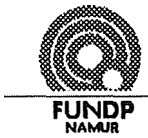
MASTER IN COMPUTER SCIENCE

Improving the generecity of an abstract interpretation algorithm through object oriented design

Delvaux, Pierre ; Englembert, Nicolas

Award date:
1996

Awarding institution:
University of Namur



Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 1995-1996

IMPROVING THE GENERICITY OF AN ABSTRACT INTERPRETATION ALGORITHM THROUGH OBJECT ORIENTED DESIGN

Pierre Delvaux & Nicolas Englebert

Promoteur: Baudouin Le Charlier

Mémoire présenté en vue de l'obtention du grade
de Licencié et Maître en Informatique

UBS 6843618

307324

Abstract

Abstract interpretation is a methodology to analyze programs statically. Abstract interpretation of Prolog is currently a very attractive field of research. Because of this, many implementations are available for specific interpretations. This report proposes a design to improve the genericity of a Prolog abstract interpreter through object oriented techniques. The result is summed up in a C++ program able to juggle a multitude of abstract domains and algorithms. Moreover, this system is built to be extended in the future.

Résumé

L'interprétation abstraite est une méthode d'analyse statique des programmes. L'interprétation abstraite de Prolog est actuellement un champs de recherche très actif. De ce fait, beaucoup d'implémentations sont disponibles pour des interprétations spécifiques. Ce mémoire propose un design pour augmenter la généricité d'un interpréteur abstrait de Prolog au moyen de techniques orientées objets. Le résultat consiste en un programme écrit en C++ capable de jongler avec une multitude de domaines abstraits et d'algorithmes. De plus, ce système est prévu pour être enrichi dans le futur.

Acknowledgments

We would like to thank all the people who contributed to this report. We especially want to thank Pascal Van Hentenryck, who invited us to Brown University, for his support as well his numerous and perceptive comments and criticisms. They have had a major impact on our work and on this report. We also thank Baudoin Le Charlier who supported the project in important ways and gave us the opportunity to fulfill our task.

Special thanks to our friend Carrie for her assistance with our English grammar.

We also thank the G.C.B. for its sympathetic and supportive atmosphere.

Table of contents

Introduction

The application field of logic programming is growing and is no longer limited to artificial intelligence. Declarative programming attracts a large number of programmers through its natural power to express facts and the relations between them.

However, this gain in expression has to be paid for by a loss of efficiency. Fortunately, static analysis, by means of abstract interpretation, provides an attractive tool to help the programmer in proving a program's properties and generating a more efficient compiled code.

Abstract interpretation of logic programs has inspired many theoretical works, as well as practical applications. Some of these ideas have been implemented but, due to these selective implementations, their focus has been limited to what is the current novelty. One regrets the loss of time and efficiency when switching from one to another to take advantage of each one's specificity. A lot of time is also wasted when re-coding common parts of the systems. Moreover, some smart combinations of abstract interpretation algorithms could result in greater benefits than a series of lone executions.

The assignment is therefore to integrate some Prolog abstract interpreters into a coherent global design, carried out under the supervision of Baudouin Le Charlier and Pascal Van Hentenryck. The aggregated system must improve the genericity of Prolog abstract interpretation by providing to the user a single interface to the different features. This design should also respect the criteria of handling ease and ability for future extensions.

As the framework is made of different inter-operating components, a special attention should be given to their interfaces. The object oriented programming paradigm, by providing opportunities of encapsulation and genericity, is a suitable foundation to achieve our objectives.

The first part of this report focuses on a review of logic programming, abstract interpretation and object oriented programming. It is followed by the core section where the design of the different components is exposed. Finally, the results of the work are presented and evaluated.

Part I: Background

This first part has the purpose of familiarizing the reader with the core concepts present in this report. As the target language of the static analysis is Prolog, we first review what logic programming is. Secondly, the properties of Prolog programs are deduced from abstract interpretation; we thus explain this concept as well as general fixpoint computation algorithms. Finally, while genericity is achieved by an object oriented paradigm and the code is implemented in C++, we tackle these notions.

We tried to find a balance between completeness and comprehensibility. We illustrate each subject with several examples.

1. PROLOG AND LOGIC PROGRAMMING

This introduction to Prolog and logic programming is there to insure that we share the same basic knowledge. We expose the key concepts of logic programming and broaden them to the Prolog language. We thus explain the main Prolog data objects and how it manipulates them to achieve the computation of a program.

Contents of this chapter:

1.1. LOGIC PROGRAMMING AND THE PROLOG ACHIEVEMENT

As opposed to logic thinking, which finds its origin in scientific reasoning, Prolog is not so old; work on Prolog began in the early seventies. The first implementation of Prolog was released in 1972 by Alain Colmerauer and Phillippe Roussel. We think that Prolog has still not reached its maturity; that is to say its lack of efficiency in some cases prevents it from being accepted as a real software development tool by the industry.

As the name "Prolog[1]" suggests, this language tries to be an implementation of logic programming. Logic programming languages are an alternative to imperative programming languages such as C, Pascal, FORTRAN or COBOL. Logic programming languages are *high-level* and *declarative* languages.

A programming language is a system of notation for describing computations. A useful programming language must therefore be suitable both for describing (i.e., for human writers and readers of programs), and for computation (i.e., for efficient implementation on computers). But human beings and computers are so different that it is difficult to find notational devices that are well suited to the capabilities of both. Languages that favor humans are termed high-level, and those oriented for machines low-level.

Declarative programming describes *what* is computed and not *how* it is done. This kind of programming language tries to separate the logic from the control. The idea is to write a program as the specification of the solution to the given problem and then provide that text to the computer, which will be able to find out the results[2].

In logic programming, the main idea is that deduction can be viewed as a form of computation, and that the statement `P if Q and R and S` can also be interpreted procedurally as "to solve `P`, solve `Q` and `R` and `S`". Under these assumptions, a logic program is a set of axioms, or rules, defining relations between objects using a subset of first order logic; and a computation is a deduction of consequences of the program.

---

[1] Prolog means *"programmation en logique"*.

[2] We find the same idea in other programming paradigms such as functional programming.

More formally, a logic program is a finite set of clauses. A clause (or rule) is a logical sentence of the form $A \leftarrow B_1, \ldots, B_n \ (n \geq 0)$ and is read "A is implied by the conjunction of the B_i".

But Prolog is only an approximate implementation of the logic programming model on a sequential machine. In effect, when deducing a logical formula, there is no particular order between the rules or between the components of a rule. This non-determinism deduction has to be raised when implementing a programming language. A Prolog program is thus a logic program in which an order is defined both for the clauses and the atoms of the clauses.

1.2. PROLOG ABSTRACT SYNTAX

Before going further, it is important for the reader to know the structure of a Prolog program as well as the objects allowed to build it. To have a common basis for the description of the examples, we give an abstract syntax of Prolog using the BNF notations[3].

---

[3] Here is a summary recapitulating the BNF notations we used to describe Prolog abstract syntax.

<xxxxxxx>	denotes the defined object or an object used to define another.
[<xxxxxxx>]	indicates that <xxxxxxx> is optional.
{<xxxxxxx>}	means that <xxxxxxx> can be repeated n times ($0 \leq n \leq +\infty$).
|	denotes the disjunction,
::=	is used to defined an object.

```
<Program>::= { <Procedure> }
<Procedure>::= { <Clause> } (clauses have the same name)
<Clause>::= <Procedure_head> [:- <Goal> ].
<Procedure_head>::= <Predicate> [ ( {<Term>} ) ]
<Goal>::= { <Literal> }
<Literal>::= <Atom> | ¬<Atom>
<Atom>::= <Predicate>  [ ( {<Term>} ) ]
               | <Term>₁ = <Term>₂
<Term>::= <Constant> | <Variable>
               | <Predicate>  [ ( {<Term>} ) ]

<Constant> ∈ Cst    : set of constants.
<Variable> ∈ Var    : set of variables.
<Predicate> ∈ Predₙ : set of n-ary predicates (n ∈ ℕ).
```

Note also that in the following, constants and predicates are denoted by identifiers beginning with a lower-case letter while variables' identifiers begin with a capital letter.

1.3. PROLOG EXECUTION MODEL

Before discussing abstract interpretation of Prolog, we think it is important to review the concrete execution of a Prolog query. A query is a conjunction of the form $\leftarrow B_1, \ldots, B_n$ $(n > 0)$ where B_i are goals.

In Prolog, binding of variables is made by a process called *substitution.* A substitution can be viewed as an automorphism on the terms and can be depicted as a finite set of such associations where the first component of the couple is a variable and the second a term. For example:

$$\theta = \{X_1/t_1, \ldots, X_n/t_n\}$$

is a substitution and the term $S\theta$ denotes the result of simultaneously replacing in the term S each occurrence of the variable X_i by t_i. The term $S\theta$ is called an instance of S.

A computation of a Prolog program P, given a query Q, returns a set of substitutions possibly empty in which for each substitution θ there is a clause C in P such as $Q\theta \leftarrow C$.

The key concept in Prolog is the notion of *unification.* Unifying two terms is finding a substitution which, if applied to both of them, makes them syntactically identical. If such a substitution exists then it is called the *unifier* and the two terms are said to be *unifiable.* For example, the terms `f(X, g(Y))` and `f(a, Z)` are unifiable by the unifier θ = {X/a,

Z/g(Y)}. On the other hand, the terms f(X, g(Y)) and f(Z, h(W)) are not.

As explained earlier, the execution of a Prolog program is totally deterministic and there is thus an order for evaluating the clauses and the goals. When solving a goal (i.e., a query or an atom of a clause), the first literal to be solved is the leftmost one and then the next until the empty literal is reached. Given a literal to solve, the first clause whose head unifies with it is chosen and then the next until there are no more clauses satisfying the unification. When that happens, the execution backtracks to the last literal chosen, i.e., asks the last literal if it can produce another solution.

Fig. 1-1 shows an example of backtracking for the Prolog program:

```
p:- a1, a2, a3.
```

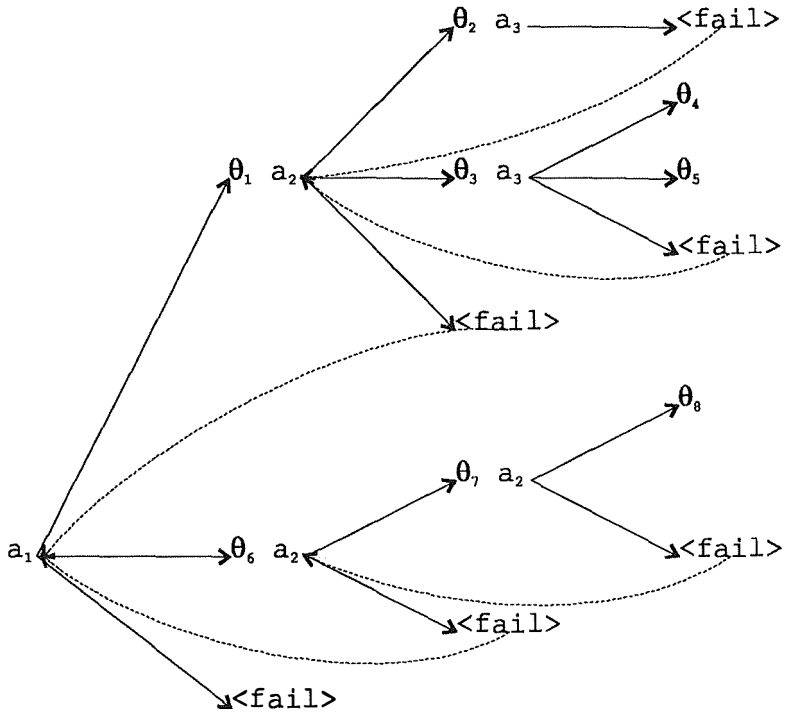


Fig. 1-1: Backtracking.

The literal a_1 produces a first solution θ_1 and suspends its execution to let the literal a_2 compute. Given the input substitution θ_1, the literal a_2 produces a solution θ_2 and suspends. The control is given to the literal a_3 which is unable to produce a result for that input. The execution then backtracks and gives the control to a_2. This last produces a second solution θ_3 which, given to a_3, permits it to generate two final solutions: θ_4 and θ_5. After the failure of a_3, the execution backtracks again just to notice that a_2

has nothing more to produce, and so backtracks one more time to gives the control to a_1. This literal produces a second solution θ_6 which is forwarded to a_2 (a new execution of this literal is thus started). This leads finally to a solution θ_8. After a few more backtrackings, the execution of the program terminates when the literal a_1 fails.

To sum up, the solutions of this program are θ_4, θ_5 and θ_8.

1.4. EXAMPLE

To clarify all the notions explained in this section, here is a small example. Let us consider the relation existing between two lists `L1` and `L2` and a third one `LR` which is their concatenation. We can express it

```
LR = L1 <> L2,
```

where <> denotes the list concatenation, or

```
append(L1,L2,LR).
```

Using the first order predicate logic and normalizing the formula in a disjunction of conjunctions, we can write it[4]

```
(L1 = [] ∧ L2 = LR) ∨ (L1 = [H|T] ∧ LR = [H|R] ∧
                       append(T,L2,R)).
```

where | denotes the addition of a first element in a list.

The associated Prolog program could be:

```
append([],L2,L2).
append([H|T],L2,[H|R]):- append(T,L2,R).
```

The execution of the query ← `append([a,b],[c,d],LR)` is now depicted in Fig. 1-2. The number on the arrows denotes the number of the clause and the unifying substitution is indicated when successful. Note also that at each recursive call, the variables are renamed to avoid conflict.

---

[4] Note that the quantifiers are omitted.

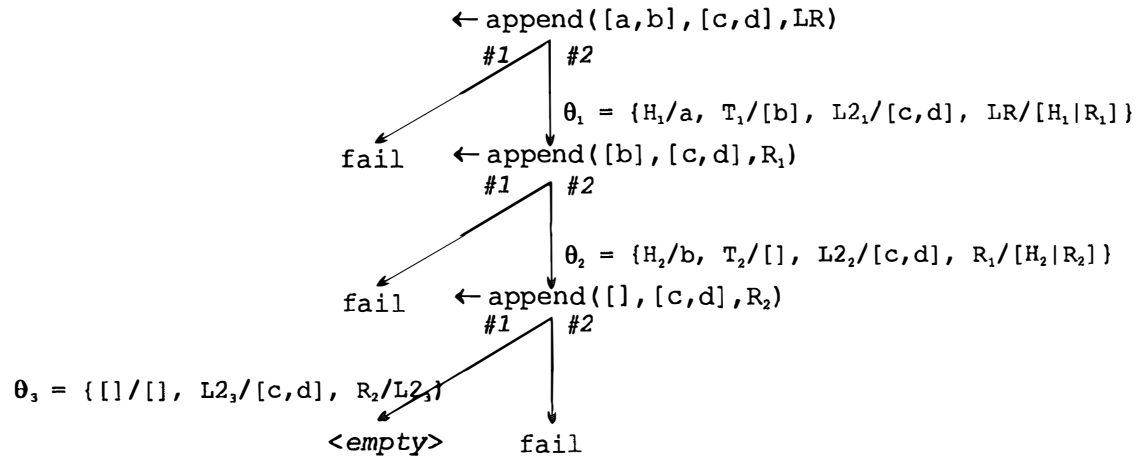


Fig. 1-2.: Execution of `append([a,b],[c,d],LR)`.

The solution to the query is given by θ which is the composition[5] of substitutions θ_1, θ_2 and θ_3 restricted to the variables present in the query. The resulting substitution θ tells us that `LR = [a,b,c,d]`.

1.5. THE CUT SYSTEM PREDICATE

We do not want to get into the subtleties of Prolog but we think it is important to explain briefly what the cut system predicate is (noted `!`). This predicate affects the procedural behavior of Prolog deduction. When executed, a cut has two effects:

- when the execution backtracks, it will not go before this literal;
- the next clauses of the current procedure will not be tried.

Let us illustrate the cut on the following example. We define the Prolog program `min/3`, which computes `Z` as the minimum of `X` and `Y`, as:

```
min(X,Y,Z):- X ≥ Y, Z = Y.
min(X,Y,Z):- Y ≥ X, Z = X.
```

The result of the query `min(5,5,Z)` gives two times the same substitution: `{Z/5}`.

We modify the program by adding a cut in the first clause:

---

[5] Because of the functional definition of the substitution, the composition here can be the usual functional composition.

```
min(X,Y,Z):- X ≥ Y, !, Z = Y.
min(X,Y,Z):- Y ≥ X, Z = X.
```

Now the result of the query `min(5,5,Z)` with this modified program gives only one substitution `{Z/5}`; the second clause is not tried because a cut has been executed in the first clause.

1.6. SUBSTITUTIONS

We must now give a more formal definition of the Prolog result of a computation, that is to say the substitutions. We first define the substitution in itself, and then how Prolog handles it as a result.

1.6.1. CONCRETE SUBSTITUTIONS

Substitutions are one of the main objects of the Prolog concrete semantics. A concrete substitution S is a finite set CS (possibly empty) of the form

$$\{X_1/t_1, \ldots, X_n/t_n\},$$

where each X_i is a variable, each t_i is a term distinct from X_i and the variables $X_1, \ldots, X_n$ are distinct. Its domain $dom(S)$ is $D = \{X_1, \ldots, X_n\}$.

1.6.2. SEQUENCE OF SUBSTITUTIONS

In general, a literal may fail or produce one or several solutions. Thus a clause or a procedure may produce several solutions.

For a given input substitution S and predicate p, the execution of a program produces a sequence of substitutions $\langle S_1, \ldots, S_n \rangle$. This is denoted by $\langle S,p \rangle \rightarrow \langle S_1, \ldots, S_n \rangle$. Moreover, the execution is described by attaching a sequence of substitutions to each program point of the procedure.

To sum up, a Prolog procedure may either:

- terminate after producing a finite number of solutions;
- produce an infinite number of solutions;
- enter an infinite loop after producing a finite number solutions.

As a consequence, a Prolog program could produce any of the sequences of substitutions depicted in Table 1-1.

Type	representation	example of such a query
empty	<>	← append([a,b],X,[c,|Y])
finite	<S_1, ..., S_n>	← append(X,Y,[a,b,c])
infinite	<S_1, ..., S_i, ...>	← append(X,[],Y)
incomplete	<S_1, ..., S_i, ⊥>	← append(X,X,X)

Table 1-1: Sequences of substitutions' typology.

The empty sequence is only a particular case of finite sequence.

1.7. REFERENCES

[BY95/9] BYTE, A Brief History of Programming Languages , September 1995.

[LEC2L1] LE CHARLIER B., lecture notes: *Logic programming*, 2° licence, 1994.

[TEN] TENNENT R. D., Principles of Programming languages, Prentice Hall.

2. ABSTRACT INTERPRETATION

The abstract interpretation algorithm is the core engine of our project. So we think that the time has come to abstract the concrete. In this chapter, we explain the notions of static analysis and abstract interpretation in general. The concepts of fixpoint, fixpoint computation and abstract domains are exposed. We then particularize these concepts to the specific case of Prolog. Finally, we consider some benefits of Prolog abstract interpretation.

Sections 2.1 and 2.2 of this chapter are based on [LECH91] and [LEVA94], respectively.

Contents of this chapter:

2.1. STATIC ANALYSIS AND ABSTRACT INTERPRETATION

2.1.1. INTRODUCTION

The basic idea behind abstract interpretation is to approximate program properties by using an abstract domain instead of the actual domain of computation.

For instance, the actual domain consisting of the integers can be replaced by the abstract domain $\{-, 0, +\}$ representing the set of negative integers, zero, and the set of positive integers. This way, properties of the sign of an expression can be found without actually computing this expression. The basic operations of the language and/or its semantics can then be associated with operations on the abstract domain which approximate them in a consistent manner (for instance the addition has to be redefined on the set $\{-, 0, +\}$).

2.1.2. REVIEW

We now introduce some definitions that are useful for the rest of this section.

- *Relation*: a relation R on a set S is a subset $S \times S$. As we prefer to use the infix notation, we write xRy for $(x,y) \in R$.
- *Partial order*: a relation R on a set S is partial order if it respects the following properties:
  1. reflexivity: $\forall\ x \in S,\ xRx$;
  2. anti-symmetry: $\forall\ x, y \in S,\ xRy \wedge yRx \Rightarrow x = y$;
  3. transitivity: $\forall\ x, y, z \in S,\ xRy \wedge yRz \Rightarrow xRz$.
- *Upper bound*: Let S be a set with a partial order $\leq$; then $x \in S$ is an upper bound of a subset $U \subseteq S$ if $u \leq x\ \forall\ u \in U$.
- *Least upper bound* (`lub` for short): Let S be a set with a partial order $\leq$; then $x \in S$ is the least upper bound of a subset $U \subseteq S$ if x is an upper bound of U and, for all upper bounds x' of U, we have $x \leq x'$.
  If it exists, the least upper bound of a set is unique.
- *Lower bound*: Let S be a set with a partial order $\leq$; then $x \in S$ is an lower bound of a subset $U \subseteq S$ if $x \leq u\ \forall\ u \in U$.
- *Greatest lower bound* (`glb` for short): Let S be a set with a partial order $\leq$; then $x \in S$ is the greatest lower bound of a subset $U \subseteq S$ if x is a lower bound of U and, for all lower bounds x' of U, we have $x' \leq x$.
  If it exists, the greatest lower bound of a set is unique.

- *Chain*: a chain of the partial order $(S,\leq)$ is a finite increasing sequence $x_0 \leq x_1 \leq \ldots \leq x_i \leq \ldots$.
- *Complete partial order* (cpo for short): the partial order $(S,\leq)$ is a complete partial order if the set S owns a minimal element, named bottom and denoted $\perp_S$, and if any chain $x_0 \leq x_1 \leq \ldots \leq x_i \leq \ldots$ of S has a least upper bound denoted $\bigcup_{i=0}^{\infty} x_i$.
- *Lattice*: a lattice is a partially ordered set S in which any two elements x_i and x_j have a least upper bound and a greatest lower bound in S.
- *Complete lattice*: a lattice L is a complete lattice if $lub(U)$ and $glb(U)$ exist for every subset $U \subseteq L$.
  The Fig. 2-1 is an example of a complete lattice.
- *Monotonic*: Let L_1, L_2 be complete lattices and $T: L_1 \rightarrow L_2$ be a mapping. T is monotonic iff $T(x) \leq T(y)$ whenever $x \leq y$, $\forall\ x \in L_1$ and $y \in L_2$.
- *Concretization function*: this function maps the abstract properties to the concrete values

  $Cc: AS \rightarrow \mathcal{P}(CS)$

  $a \rightarrow \{c: c \text{ verifies } a\}$
- *Abstraction function*: this function maps the concrete values to the abstract properties

  $Abs: CS \rightarrow AS$

  $c \rightarrow a$ a is an abstraction of c.

2.1.3. MATHEMATICAL BACKGROUND

2.1.3.1. CONCRETE COMPUTATION

We denote D the domain of values handled by a language we want to analyze by means of abstract interpretation. In general, D has a complex structure to take into account the different types of the language: scalars, structured objects, files ... We assume that D is unstructured; this does not change the nature of results but simplifies the notations. Note that we will make other simplifications without ever mentioning them. As we will see later, all this can be extended to the case of actual languages (Prolog of course) but not without complications.

Suppose we want to analyze the following procedure[6] of our language.

```
f(x) = if x > 100
       then x - 10
       else f(f(x + 11)).
```

This function[7] computes values from $\mathbb{Z}$ to $\mathbb{Z}$ (the set of integers). We are thus interested in some properties of the function computed by the procedure and an interesting one is which values could take the variables at different points of the execution.

To capture such multiple information, we have to replace individual values of D by sets of values; that is to say elements of $\wp$(D). This set is denoted C (for concrete domain, of all the possible properties). The procedure computing over D can now be replaced by a procedure computing over C (X denotes a set of integers):

$$f(X) = \{x - 10 : x > 100 \wedge x \in X\} \cup f(f(\{x + 11 : x \leq 100 \wedge x \in X\})). \quad (1)$$

Basic operations on individual values can generally be replaced by operations handling sets of values. Procedures modified this way compute the set of possible results corresponding to the set of possible inputs. Actually, those modified procedures are not useful for two reasons. The first reason leads us to the notion of abstract domain and the second to the notion of the least fixpoint of a transformation.

2.1.3.2. ABSTRACT DOMAINS

Not all sets of values are workable; it is, moreover, theoretically impossible. The concrete domain C is thus replaced by an abstract domain A conserving only some elements of C such that any element of C can be approximated by an element of A.

Technically, it is often demanded that A be a complete lattice or a complete partial order and that two monotonic functions $Abs: C \rightarrow A$ and $Cc: A \rightarrow C$ exist and verify the two following conditions:

- $\forall c \in C: Cc(Abs(c)) \supseteq c$;
- $\forall a \in A: Abs(Cc(a)) = a$.

---

[6] We use functional notations.

[7] This function is called the "91-function".

That is to say, the abstraction function Abs associates each set of values with its best approximation and the concretization function Cc associates each element of A with the set of values it represents.

A simple way to design an abstract domain is to define a partition $\{S_1, \ldots, S_n\}$ of C, an arbitrary set $A = \{a_1, \ldots, a_n\}$ and to define the concretization function $Cc: A \rightarrow \mathcal{P}(C)$, by

$$Cc(a_i) = S_i \qquad 1 \leq i \leq n.$$

We call these abstract domains *flat domains* since their elements are not comparable.

An example of such a domain is the classical domain for sign analysis defined as:

$C = \mathbb{Z}$,
$A = \{-, 0, +\}$,
$Cc(-) = \{i \mid i < 0\}$,
$Cc(0) = \{0\}$,
$Cc(+) = \{i \mid i > 0\}$.

We can complete a flat domain such as each pair of abstract values has an upper bound. The ordering is introduced by the concretization relation. Note that it is not absolutely required to have a unique least upper bound, although this additional requirement is natural.

Once upper bound has been added, one may remove some original elements because they can be approximated by one of the upper bounds. The idea is to keep only elements that bring "interesting" information. Hence there are many ways to complete a flat domain A.

The most complete is $\mathcal{P}(A)$ with $Cc: \mathcal{P}(A) \rightarrow \mathcal{P}(C)$ defined by

$$Cc(\{a_{i,1}, \ldots, a_{i,j}\}) = \bigcup_{k=1}^{j} Cc(a_{i,k}).$$

Such a completed domain is called a *power set domain.* Obviously if n is the number of elements of A, $\mathcal{P}(A)$ contains 2^n elements. Therefore power set domains can only be used for small values of n.

An other systematic way to complete a flat domain is to only add one "top" element $\top$ with $Cc(\top) = C$. Notice that $\{\top\}$ is itself a completed (albeit not very interesting) domain.

The power set domain for sign analysis is defined as:

$C = \mathbb{Z}$,
$A = \{\bot, -, 0, +, \leq, \neq, \geq, \top\}$,
$Cc(\bot) = \{\}$,
$Cc(-) = \{i \mid i < 0\}$,
$Cc(0) = \{0\}$,
$Cc(+) = \{i \mid i > 0\}$,
$Cc(\leq) = \{i \mid i \leq 0\}$,
$Cc(\neq) = \{i \mid i \neq 0\}$,
$Cc(\geq) = \{i \mid i \geq 0\}$,
$Cc(\top) = \mathbb{Z}$.

The Fig. 2-1 depicts the order between elements in a Hasse diagram. A Hasse diagram is made up of nodes and edges in such a way that there is an edge between nodes `a` and `b` iff $a \leq b$ and there is no `c` such that $a \leq c$ and $c \leq b$.

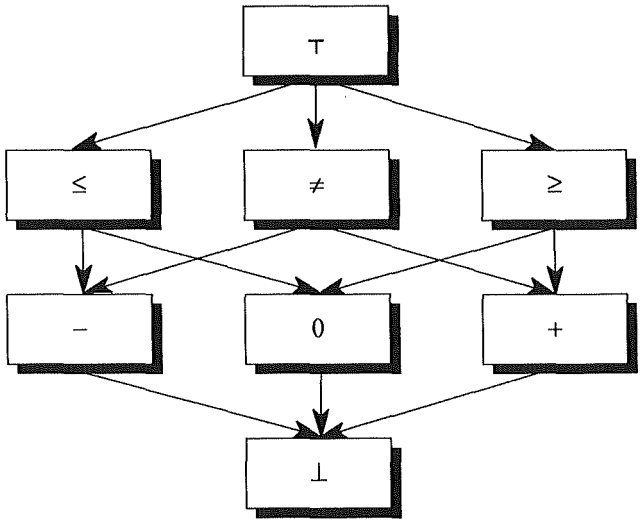


Fig. 2-1: Hasse diagram for sign analysis.

In the procedure (1), $C = \mathcal{P}(\mathbb{Z})$. An example of abstract domain `A` would be the set $\mathbb{I}$ of intervals. An interval `[i..s]`, where $i, s \in \mathbb{Z} \cup \{-\infty, +\infty\}$, is the set of integers `e` such that $i \leq e \leq s$. The set $\mathbb{I}$ is a complete lattice for the set inclusion $\subseteq$. `{}` is the smallest element (bottom) and `[-∞..+∞]` the greatest (top). `Abs(X)` is the interval `[min(X)..max(X)]`. `Cc(X)` is the inclusion of $\mathbb{I}$ in $\mathcal{P}(\mathbb{Z})$.

It is possible to rewrite the former procedure so that it maps an interval of inputs to an interval of results.

$$f([i..s]) = [max(91, i - 10)..(s - 10)] \amalg f(f([(i + 11)..min(s + 11, 111)])). \quad (2)$$

where $\amalg$ approximates the union of two intervals by the smallest interval which includes both of them. From this definition we try to approximate the set of values produced by the function. We get the two following equations:

$$f([-\infty..+\infty]) = [91..+\infty] \amalg f(f([-\infty..111])),$$
$$f([-\infty..111]) = [91..101] \amalg f(f([-\infty..111])).$$

These equations demonstrate that the usual computation method is impossible: it generates an infinite loop since $f([-\infty..111])$ recursively triggers off the computation of itself. We just pointed out the second difficulty. Computation over abstract domains cannot blindly simulate computation over the standard domain. We separate the difficulty into two levels. First, we give an accurate (mathematical) meaning to the procedures such as (1) and (2), thanks to the notion of the least fixpoint of monotonic transformation. We then expose the problem of computing least fixpoints.

2.1.3.3. LEAST FIXPOINT OF A TRANSFORMATION

We reason over the abstract domain of intervals $\mathbb{I}$, but the same process could be applied over any abstract domain. Let $\mathbb{I} \to \mathbb{I}$ be the set of monotonic and continuous functions from $\mathbb{I}$ to $\mathbb{I}$; that is to say, such that:

- $\forall I, I' \in \mathbb{I}: I \subseteq I' \Rightarrow f(I) \subseteq f(I')$;
- for all finite series of embedded intervals $I_1 \subseteq I_2 \subseteq \ldots \subseteq I_i \subseteq \ldots$ we have

$$f\left(\bigcup_{i=0}^{\infty} I_i\right) = \bigcup_{i=0}^{\infty} f(I_i).$$

These conditions express that f is really an abstraction of a function from $\mathbb{Z}$ to $\mathbb{Z}$. The set $\mathbb{I} \to \mathbb{I}$ can be endowed with an order:

$$f \leq g \text{ iff } \forall I \in \mathbb{I}: f(I) \subseteq g(I).$$

This order relation means that the procedure corresponding to g produces at least as many results as the procedure corresponding to f. The definition (2) can be replaced by a transformation of function:

$$\tau: (\mathbb{I} \to \mathbb{I}) \to (\mathbb{I} \to \mathbb{I}).$$

with

$$(\tau\ f)\ ([i..s]) = [max(91, i - 10)..(s - 10)] \amalg f(f([(i + 11)..min(s + 11, 111)])).$$

Hence, the equation (2) simply means that the function f is a fixpoint of the transformation τ; that is to say:

$$\tau(f) = f.$$

It is possible to demonstrate the following results.

- Provided that the concrete and abstract domains verify the formerly mentioned properties, every fixpoint of τ gives a correct approximation of the properties of the associated procedure.
- The transformation τ has necessarily a least fixpoint (the most accurate).
- The least fixpoint of τ is equal to the limit of an increasing series of approximations:

$$f_0 \leq f_1 \leq \ldots \leq f_k \leq \ldots$$

where $f_0(I) = \{\} \ \forall \ I \in \mathbb{I}$,
$f_{k+1} = \tau(f_k) \ \forall \ k \geq 0$.

In our example, we can verify after a drudgery computation that

$$f_k([i..s]) = \begin{cases} [\max(91,i-10)..\max(91,s-10)] & \text{if } s \geq l(k) \\ \{\} & \text{if } s < l(k) \end{cases}$$

with

$$l(k) = \begin{cases} +\infty & \text{if } k = 0, \\ 102 - k & \text{if } 1 \leq k \leq 12, \\ 222 - 11k & \text{if } k \geq 13. \end{cases}$$

As $l(k)$ tends to $-\infty$ when k tends to $+\infty$, the series of functions converges to the function f such that

$$f([i..s]) = [\max(91,i-10)..\max(91,s-10)]. \quad (3)$$

This function is the least fixpoint of the transformation τ in our example. The interest of such a function is that the computation of a single value gives us information about an infinity of executions of the original procedure (from $\mathbb{Z}$ to $\mathbb{Z}$). We have for example

$$f([-\infty..+\infty]) = [91..+\infty],$$

which indicates that all the values produced by the standard procedure are greater or equal to 91. We can also compute that

$$f([-\infty..101]) = \{91\},$$

which means that for all input smaller or equal to `101`, the procedure returns `91` or does not terminate[8].

However, the way we proceeded to compute these results is not satisfactory for an automated processing: the determination of an explicit form of the fixpoint such as (3) can only be realized by a specific reasoning. Instead, we must find methods of fixpoint computation based only on the recursive definition (2). It is the purpose of the next section.

2.1.4. GENERAL ALGORITHMS OF FIXPOINT COMPUTATION

Fixpoint computation of some transformations associated with programs according to an abstract semantic is close to computations of recurrent procedures in a programming language. There are however two major differences:

- the computation must terminate in all possible cases;
- it is generally sufficient to compute an approximation of the fixpoint[9].

2.1.4.1. BOTTOM-UP AND TOP-DOWN EVALUATION OF RECURSIVE DEFINITIONS

Fixpoint computation algorithms are a generalization of the methods used to compute the values of functions defined recursively. That is why we first expose these methods before generalizing them for abstract interpretation. Let us consider the following recursive definition:

```
f(x) = if x ∈ {0, 1}
       then x
       else f(x - 1) + f(x - 2).
```

To compute a particular value `f(v)`, we can first evaluate the function *bottom-up*, starting with "small values" `0` and `1` for which the value of `f` is immediate, and then by progressively propagating these results to `2`, `3`, `4`, ... until we obtain the desired value. If we want to compute `f(4)`; it looks like this:

---

[8] In effect, our abstract domain does not deal with the issue of termination.

[9] Note that the fixpoint itself, in general, gives only an approximation of the actual properties.

```
f(0) = 0
f(1) = 1
f(2) = f(1) + f(0) = 1 + 0 = 1
f(3) = f(2) + f(1) = 1 + 1 = 2
f(4) = f(3) + f(2) = 2 + 1 = 3
```

We can also evaluate the function *top-down*, starting from the expression `f(v)` to compute, and by further developing it until we obtain an expression completely computable. Again, to compute `f(4)`, it looks like:

```
f(4)= f(3) + f(2)
    = (f(2) + f(1)) + (f(1) + f(0))
    = ((f(1) + f(0) + f(1)) + (f(1) + f(0))
    = ((1 + 0) + 1) + (1 + 0)
    = 3
```

The bottom-up method seems to be more efficient on the example above. However, it is difficult to systematize it because we need to find the right series of values to compute so that all the needed values are already computed. It is not always possible. The top-down method has the advantage of being systematic. Unfortunately, it is very inefficient because the same value can often be reevaluated. Above, `f(2)` is computed twice. Generally, top-down computation is exponential in time whereas bottom-up is linear.

Fortunately, the top-down method can be improved to be generally as efficient as the bottom-up method. This improvement, know as memoization, is enhanced for the abstract interpretation algorithms. The idea is to record in a table the values already computed in order to prevent them from being reevaluated. Our example, using this improvement, gives:

```
f(4) = f(3) + f(2)
     = (f(2) + f(1)) + f(2)
     = ((f(1) + f(0)) + f(1)) + f(2)
     = ((1 + f(0)) + f(1)) + f(2)    {f(1) = 1}
     = ((1 + 0) + f(1)) + f(2)       {f(0) = 0}
     = (1 +f(1)) + f(2)              {f(2) = 1}
     = 2 + f(2)                      {f(3) = 2}
     = 3                             {f(4) = 3}
```

Now, this modified algorithm is linear since every expression `f(v)` is only evaluated once.

2.1.4.2. ABSTRACT INTERPRETATION ALGORITHMS

The issue is to compute the least fixpoint of the abstract transformation:

$$\tau: (A \rightarrow A) \rightarrow (A \rightarrow A).$$

Remember that we want an algorithm able to compute $f(a)$ for all abstract values a.

A. Bottom-up algorithm

The simplest way to compute such $f(a)$ value can be done by a series of approximations:

- $f_0(a) = \bot \quad \forall\ a \in A,$
- $f_{k+1} = \tau(fk) \quad \forall\ k \geq 0,$
- $f = f_n \quad$ such that $f_{n+1} = f_n.$

This bottom-up method demands that the domain A be finite[10]. Moreover, the results produced by this algorithm are sometimes less accurate with regard to the top-down.

B. Top-down algorithm

The algorithm explained here is based on the top-down method and uses an enhanced memo-ization; we store intermediate and partial results of computation. We start with the recursive definition of the fixpoint and then try to recursively compute the value $f(a)$. During the computation, we keep an up-to-date table of values a_i for which a recursive call has been already initiated (whether terminated or not) together with its associated lower approximation of $f(a_i)$. When the same recursive call is reconsidered, it is not developed but its current approximation stored in the table is returned. At each end of a call, the content of the table is updated with the last value computed. As this later is generally an approximation (since the value in the table are so), we repeat the same computation until the result cannot be further improved (i.e., the result is *stable*).

Let us illustrate this top-down algorithm with the fixpoint computation of the 91-function. We have to compute $f([-\infty..+\infty])$ for the transformation

---

[10] A widening can be use in the case of infinite domains.

$$(\tau\ f)\ ([i..s]) = [\max(91, i - 10)..(s - 10)] \amalg f(f([(i + 11)..\min(s + 11, 111)])).$$

Each time a recursive call is initiated, a value is added in the table with the associate current "approximated" result $\{\}$. Hence, when we start the computation, the approximated value of $f([-\infty..+\infty])$ in the table is $\{\}$. After the first iteration, we get

$$f([-\infty..+\infty]) = [91..+\infty] \amalg f(f([-\infty..111])).$$

Thus a call for $f([-\infty..111])$ (whose current approximation is $\{\}$) is initiated. This call can be further developed in

$$f([-\infty..111]) = [91..101] \amalg f(f([-\infty..111])).$$

The same recursive call should be initiated but we instead pick up its current approximated value in the table (i.e., $\{\}$); this would have otherwise entailed an infinite loop. So we get

$$\begin{aligned} f([-\infty..111]) &= [91..+\infty] \amalg f(\{\}) \\ &= [91..+\infty]. \end{aligned}$$

The new result for this call is used to update the table for that entry and then the same computation is reconsidered just in case we could improve the current result due to the new information just obtained.

$$\begin{aligned} f([-\infty..111]) &= [91..+\infty] \amalg f(f([-\infty..111])) \\ &= [91..+\infty] \amalg f([91..101]). \end{aligned}$$

This new iteration triggers off the computation of $f([91..101])$. Again, the same method is applied to compute the successive approximations of $f([91..101])$. Since it is quite long, it is not mentioned. The least fixpoint computation finally stabilizes and terminates with the best possible result

$$f([-\infty..+\infty]) = [91..+\infty].$$

This algorithm can be systematically implemented for any kind of programming language, provided it has been endowed with an abstract semantic allowing to associate with any program a transformation τ. This is theoretically always possible.

C. Approximation, termination and acceleration of convergence

The abstract domain $\mathcal{I}$ of intervals is infinite. With such a domain, the top-down algorithm can loop. This is the case if an infinity of different

calls are initiated. It is not the case if the same call is recursively repeated. This is avoided by the table. The previous example was not concerned with this eventuality but this is not always the case. To avoid such problems, we could limit ourselves to finite abstract domains but it is not always possible to find a better finite domain approximating the infinite domain we wish to use. A much cleverer approach consists of dynamically choosing those approximations, according to the particular example. The idea is to replace the virtually infinite set of values being considered by a finite set which covers them; that is to say we *widen*[11] it. The results obtained this way are generally safe (or consistent) approximations of the fixpoint values. The use of these approximations, moreover, enables faster convergence. We can illustrate this again with the computation of `f([-∞..+∞])`. We first compute

$$f([-\infty..+\infty]) = [91..+\infty] \amalg f(f([-\infty..111])).$$

As $[-\infty..111] \subseteq [-\infty..+\infty]$, we can replace `f([-∞..111])` by `f([-∞..+∞])` (with, of course, the risk of losing accuracy); that gives us

$$\begin{aligned} f([-\infty..+\infty]) &\cong [91..+\infty] \amalg f(f([-\infty..+\infty])) \\ &\cong [91..+\infty] \amalg f(\{\}) \\ &\cong [91..+\infty]. \end{aligned}$$

When reconsidering the computation, we finally get

$$\begin{aligned} f([-\infty..+\infty]) &\cong [91..+\infty] \amalg f(f([-\infty..+\infty])) \\ &\cong [91..+\infty] \amalg f([91..+\infty]) \\ &\cong [91..+\infty] \amalg f([-\infty..+\infty]) \\ &\cong [91..+\infty] \amalg [91..+\infty] \\ &\cong [91..+\infty]. \end{aligned}$$

The computation stops after two iterations with the best possible result, while the "exact" algorithm would have required lots of iterations. Our example is well tailored for the account. Unfortunately, it is possible to find less satisfying ones. Let us compute, for example, `f([-∞..100])` and replace the recursive calls by a widened call. We get:

$$\begin{aligned} f([-\infty..100]) &\cong \{\} \amalg f(f([-\infty..111])) \\ f([-\infty..111]) &\cong [91..101] \amalg f(f([-\infty..111])) \\ &\cong [91..101]. \end{aligned}$$

[11] This operation presupposes the existence of an underlying least upper bound function.

So,

```
f([-∞..100]) ≅ f([91..101])
             ≅ f([-∞..111])
             ≅ [91..101].
```

The result is less accurate than previously but obtained much more quickly. The design of operations allowing the convergence to speed up without losing too much precision is an intricate task, depending heavily on the specific abstract domain.

D. Monovariant and polyvariant algorithms

When computing an abstract interpretation, a decision must be made about keeping track of the input and output values. We can either store all values or "lump" them. The second case implies in general that the underlying domain be complete. We respectively call them *polyvariant* and *monovariant* algorithms.

In general, monovariant algorithms imply a loss of accuracy because of the upper bound operation.

For instance, in the previous example, when computing `f([-∞..+∞])`, a polyvariant algorithm would keep track of two recursive calls, `f([-∞..111])` and `f([91..101])`. On the other hand, a monovariant algorithm would only remember `f([-∞..+∞])`.

2.2. ABSTRACT INTERPRETATION OF PROLOG PROGRAMS

2.2.1. INTRODUCTION

We now particularize the above notions to the case of the abstract interpretation of Prolog programs.

In this section, we first explain the way we transform Prolog programs to make them more convenient to handle. We then introduce a typology for Prolog terms and the abstract operations needed for abstract interpretation. Finally, we discuss different types of abstract interpretation algorithms and consider the different benefits they bring.

2.2.2. NORMALIZED PROGRAMS

As we do not want to miss any operations, for example unification in the head, we have to *normalize* the input Prolog program before any other

processing. As we will see, a normalized program is also more convenient to handle.

Given a Prolog program, we normalize it with the help of an ordered set of variables $\{X_1, \ldots, X_n, \ldots\}$. These variables are called program variables. A normalized program is a set of clauses

$$p(X_1, \ldots, X_n) :- l_1, \ldots, l_r$$

where $p(X_1, \ldots, X_n)$ is called the head, and $l_1, \ldots, l_r$ the body. If a clause contains m variables, these variables are necessarily $X_1, \ldots, X_m$. The literal in the body of the clause are of the form

- $q(X_{i1}, \ldots, X_{im})$ where $X_{i1}, \ldots, X_{im}$ are distinct variables;
- $X_{i1} = X_{i2}$ with $X_{i1} \neq X_{i2}$;
- $X_{i1} = f(X_{i2}, \ldots, X_{in})$ where f is a function of arity $n-1$ and $X_{i2}, \ldots, X_{in}$ are distinct variables.

For instance, the following text of `append/3` is the normalized version of the program defined above:

```
append(X1,X2,X3):-
        X1 = [],
        X2 = X3.
append(X1,X2,X3):-
        X1 = [X4|X5],
        X3 = [X4|X6],
        append(X5,X2,X6).
```

The translation process from a raw Prolog program into its normalized version is automatic. The advantage of normalized programs is that an abstract substitution for a procedure `p/n` is always expressed in terms of variables $X_1, \ldots, X_n$. This greatly simplifies all the traditional problems encountered with renaming.

2.2.3. INSTANCIATION DEGREE OF A TERM ~ MODES

In abstract interpretation, we are often interested in the instanciation level of a term. A term can be a constant, a variable or made of term(s) structured by the mean of a function. We now explore the *groundness* of the term.

If a term contains only constants, it is called *ground*. On the other hand, if a term is only made of variables, its mode is called *var*. Between these two extremes, we can define several variations:

- *nor ground nor var*: partially instanciated,
- *ground or var*: not partially instanciated,
- *not ground*,
- *not var*,
- *any*: any term.

2.2.4. ABSTRACT OPERATIONS ON THE DOMAINS

In this section, we explain the abstract operations that are needed by the abstract interpretation algorithms to handle the substitutions and thus perform their job properly. We have two types of abstract operations: some operations are abstract versions of concrete operations which are performed by a Prolog compiler to extract the semantic of a Prolog program, whereas the others are peculiar to the abstract interpretation algorithms.

We define the following operations on the abstract domains:

- UNION(β_1, β_2): where β_1 and β_2 are abstract substitutions on the same domain; this operation returns an abstract substitution representing all the substitutions satisfying at least β_1 or β_2. It is used to compute the output of a procedure given the outputs for its clauses. More formally it is the least upper bound of the cpo's elements β_1 and β_2.
- AI_VAR(β): where β is an abstract substitution on $\{X_1, X_2\}$; this operation returns the abstract substitution obtained from β by unifying variables X_1 and X_2.
- AI_FUNC(β, f): where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and f is a predicate symbol of arity n−1; this operation returns the abstract substitution obtained from β by unifying X_1 and $f(X_2, \ldots, X_n)$.
- EXTC(c, β): where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and c is a clause containing variables $\{X_1, \ldots, X_m\}$ ($m \geq n$); this operation returns the abstract substitution obtained by extending β to accommodate the new free variables of the clause. It is used at the entry of a clause to include the variables in the body not present in the head.
- RESTRC(c, β): where β is an abstract substitution on the clause variables $\{X_1, \ldots, X_m\}$ and $\{X_1, \ldots, X_n\}$ are the head variables of clause c ($m \geq n$); this operation returns the abstract substitution obtained by projecting β on variables $\{X_1, \ldots, X_n\}$. It is used at the exit of a clause to restrict the substitution to the head variables only.
- RESTRG(l, β): where β is an abstract substitution on $\{X_1, \ldots, X_n\}$, and l is a literal $p(Xi_1, \ldots, Xi_m)$ or $Xi_1 = Xi_2$ or $Xi_1 = f(Xi_2, \ldots, Xi_m)$; this operation returns the abstract substitution obtained by:

1. projecting β on $(Xi_1, \ldots, Xi_m)$ obtaining β′;
2. expressing β′ in terms of $\{X_1, \ldots, X_m\}$ by mapping Xi_k to X_k.

It is used before the execution of a literal in the goal of a clause. The resulting abstract substitution is expressed in terms of $\{X_1, \ldots, X_m\}$, as an input abstract substitution for p/n.

- EXTG(l, β, β′): where β is an abstract substitution on $D = \{X_1, \ldots, X_n\}$, the variables of the clause where l appears, l is a literal
  - $p(Xi_1, \ldots, Xi_m)$
  - or $Xi_1 = Xi_2$
  - or $Xi_1 = f(Xi_2, \ldots, Xi_m)$ with $(Xi_1, \ldots, Xi_m) \subseteq D$

  and β′ is an abstract substitution on $(X_1, \ldots, X_m)$ representing the result of $p(X_1, \ldots, X_m)$ β" where β" = RESTRG(l, β); this operation returns the abstract substitution obtained by instantiating (abstractly) β to take into account the result β′ of the literal l. It is used after the execution of a literal to propagate the results of the literal to all variables of the clause.

2.2.5. ABSTRACT INTERPRETATION ALGORITHMS

In this section, we discuss the engine of a Prolog abstract interpretation algorithm: the computation of the least fixpoint. We explain how we keep track of the interesting results of the computation (i.e., the set of abstract tuples). Another important part of the conception of an abstract interpretation algorithm is to avoid redundant computations; this point is thus tackled too. After these discussions, the generic algorithms are exposed. A big part of this section is based on [LEVA94].

2.2.5.1. ABSTRACT SEMANTICS

The abstract semantics are defined in terms of *abstract tuples*. An abstract tuple is of the form: $(\beta_{in}, p, \beta_{out})$ where p is a predicate of arity n and β_{in} and β_{out} are abstract substitutions on variables $X_1, \ldots, X_n$.

UD is the underlying domain of the program, i.e., the set of pairs (β_{in}, p) where p is a predicate symbol of arity n and β_{in} is an abstract substitution on variables $X_1, \ldots, X_n$.

Let us denote sat, a set of abstract tuples. There exists at most one β_{out} for each pair (β_{in}, p) such that $(\beta_{in}, p, \beta_{out})$ and we indicate that substitution by $sat(\beta_{in}, p)$. A set of abstract tuples is endowed with a structure of cpo by defining:

- $\perp = \{(\beta, p, \perp_D) \mid (\beta, p) \in UD$ and is the smallest element in $(AS_D, \leq)\}$;
- $sat \leq sat' \equiv \forall\ (\beta, p) \in UD: sat(\beta, p) \leq sat'(\beta, p)$,

where AS_D is the set of abstract substitutions on the set of variables $D = \{X_1, \ldots, X_n\}$.

The abstract semantics is defined as the least fixpoint of the transformation TSAT, depicted below (Fig. 2-2). Informally speaking, the function $T_p(\beta, p, sat)$ executes all clauses defining p on substitution β and takes the union of the results. The function T_c executes one clause by extending the substitution, executing the body, and restricting the substitution. The function T_b executes the body of a procedure by considering each literal in turn. When the literal is a procedure call, a lookup in the sat is performed; otherwise the operations AI_VAR or AI_FUNC are executed. Operation RESTRG is used before calling any literal and operation EXTG is performed after each call.

$$TSAT(sat) = \{(\beta, p, \beta_2): (\beta, p) \in UD \text{ and } \beta_2 = T_p(\beta, p, sat)\}$$

$$T_p(\beta, p, sat) = UNION(\beta_1, \ldots, \beta_n)$$
where $\beta_i = T_c(\beta, c_i, sat)$,
$c_1, \ldots, c_n$ are the clauses of p.

$$T_c(\beta, c, sat) = RESTRC(c, \beta_2)$$
where $\beta_2 = T_b(EXTC(c, \beta), b, sat)$,
b is the body of c.

$$T_b(\beta, <>, sat) = \beta.$$
$$T_b(\beta, l.g, sat) = T_b(\beta_3, g, sat)$$
where $\beta_3 = EXTG(l, \beta, \beta_2)$
$\beta_2 = sat(\beta_1, p)$ if l is p(...)
$AI\_VAR(\beta_1)$ if l is $x_i = x_j$
$AI\_FUNC(\beta_1)$ if l is $x_i = f(...)$
$\beta_1 = RESTRG(l, \beta)$.

Fig. 2-2: The abstract semantics.

2.2.5.2. MANIPULATION OF THE SET OF ABSTRACT TUPLES

Mainly two operation need to be defined on the set of abstract tuples: the extend and adjust operations. Informally speaking, the EXTEND

operation is intended to extend a set of tuples with a new element while the ADJUST is intended to update the result of a pair (β, p).

The two operations can be specified as follows, assuming that the domain of a set of abstract tuples sat, denoted dom(sat), is the set of pairs (β, p) for which there exists a β' such that $(\beta, p, \beta') \in$ sat.

- EXTEND$(\beta, p,$ sat$)$, given an abstract substitution β, a predicate symbol p, and a set of abstract tuples sat which does not contain (β, p) in its domain, returns a set of abstract tuples sat' containing (β, p) in its domain. According to the implementation, the value sat'(β, p) can be defined as the least upper bound of all sat(β', p) such that $\beta' \leq \beta$ or it can be defined as $\perp$.
- ADJUST$(\beta, p, \beta',$ sat$)$, where β' represents a new result computed for the pair (β, p), returns a sat' which is sat updated with this new result. Here, the ADJUST implementation depends upon the sat implementation[12]. The minimal goal of this operation is to adjust the result of (β, p) with β' the following way: the value of sat'(β, p) is equal to lub$(\beta',$ sat$(\beta, p))$.

2.2.5.3. OVERVIEW OF THE ABSTRACT INTERPRETATION ALGORITHM

A brute-force approach to the generic abstract interpretation algorithm would be to compute the least fixpoint of TSAT entirely. We can easily understand that such an approach would involve much unnecessary work.

The purpose of the abstract interpretation algorithm is to converge toward a set of abstract tuples that includes $(\beta_{in}, p, \beta_{out}) \in$ least fixpoint of TSAT but as few other elements as possible. The algorithm computes a series of lower approximations $sat_0, \ldots, sat_n$ such that $sat_i < sat_{i+1}$ and sat_n contains $(\beta_{in}, p, \beta_{out})$. The algorithm then moves from one set to another by selecting

- an element (α, q) which is not present but needs to be computed, or
- an element (α, q) whose value $sat_i(\alpha, q)$ can be improved because the values of some elements it depends upon have been updated.

There are still many decisions to take into account, including the detection of termination and the choice of the elements to work on.

Informally, the algorithm works as follows. Given an initial pair (β_{in}, p), it executes the function T_p of the abstract semantics. At some

[12] For more information about these implementation see section II.6.1.

point, the computation may need the value of (α_{in}, q) which may not be defined or is just approximated at that stage of the computation. In this situation, the algorithm starts a new subcomputation to obtain the value of (α_{in}, q) or a lower approximation of it. This computation is carried out in the same way as the primary computation except in the case where a value for (β_{in}, p) is needed. In that case, instead of starting a new computation (that may generate an infinite loop), the algorithm simply looks up the current value of (β_{in}, p). The execution of the initial pair (β_{in}, p) is only resumed once the computation of (α_{in}, q) is completed. Note that if the computation of (α_{in}, q) has required the value of (β_{in}, p) then its resulting substitution may only be approximated and hence (α_{in}, q) has to be reconsidered if the value of (β_{in}, p) is updated. In the algorithm, a dependency graph is used to detect when an element needs to be reconsidered.

2.2.5.4. PROCEDURE CALL DEPENDENCIES

The goal of the data structure described below is to avoid redundant computations during the abstract interpretation.

Redundant computations may occur in a variety of situations. For instance, the value of a pair (α, q) may have reached its definitive value (the value of $(\alpha, q) \in$ least fixpoint of TSAT) and hence subsequent considerations of (α, q) should only look up its value instead of starting a new subcomputation. Mutually recursive programs are another important case. For those programs, we would like the algorithm to reconsider a pair (α, q) only when some elements which it is depending upon have been updated. In other words, keeping track of the procedure call dependencies may substantially improve the efficiency of some classes of programs.

A. The dependency graph

The basic intuition of the dependency graph is that $dp(\beta, p)$ represents at some point the set of pairs upon which (β, p) directly depends.

A dependency graph dp is a set of tuples of the form $<(\beta, p), lt>$, where lt is a set $\{(\alpha_1, q_1), \ldots, (\alpha_n, q_n)\}$ $(n \geq 0)$, such that, for each (β, p), there exists at most one lt such that $<(\beta, p), lt> \in dp$.

We denote by $dp(\beta, p)$ the set lt such that $<(\beta, p), lt> \in dp$ if it exists. We also denote by dom(dp) the set of all (β, p) such that $<(\beta, p), lt> \in dp$ and by codom(dp) the set of all (α, q) such that there exists a tuple $<(\beta, p), lt> \in dp$ satisfying $(\alpha, q) \in lt$.

B. Transitive closure of the dependencies

The trans_dp(β,p,dp) represents all the pairs which, if updated, would require reconsidering (β,p). (β,p) will not be reconsidered unless one of these pairs is updated.

Let dp be a dependency graph and assume that (β,p) ∈ dom(dp). The trans_dp(β,p,dp) is the smallest subset of codom(dp) closed by the following two rules:

- if (α,q) ∈ dp(β,p) then (α,q) ∈ trans_dp(β,p,dp);
- if (α,q) ∈ dp(β,p), (α,q) ∈ dom(dp), and (α',q') ∈ trans_dp(α,q,dp) then (α',q') ∈ trans_dp(β,p,dp).

C. Operations

Let us specify the three following operations:

- REMOVE_DP({(α₁,q₁),...,(αₙ,qₙ)},dp) removes from the dependency graph dp all elements <(α,q),lt> for which there is a (αᵢ,qᵢ) ∈ trans_dp(α,q,dp).
- EXT_DP(β,p,dp) inserts an element <(β,p),∅> in dp.
- ADD_DP(β,p,α,q,dp) simply updates dp to include the dependency of (β,p) with regard to (α,q). After its execution (α,q) ∈ dp(β,p).

The main intuition here is that the algorithm makes sure that the elements (β,p) that need to be reconsidered are such that (β,p) ∉ dom(dp). Conversely, elements of dom(dp) do not (as yet) require reconsideration.

2.2.5.5. TOP-DOWN ALGORITHM

We are now in a position to present the generic abstract interpretation algorithm. The algorithm is composed of three procedures and is shown in Fig. 2-3.

The top-level procedure solve which, given an input substitution β_{in} and a predicate symbol p, returns the final dependency graph and the set of abstract tuples sat containing $(\beta_{in},p,\beta_{out})$ ∈ least fixpoint of TSAT.

Procedure solve_call receives as inputs an abstract substitution β_{in}, its associated predicate symbol p, a set suspended of pairs (β,q), sat, and a dependency graph dp. The set suspended contains all pairs (β,q) for which a subcomputation has been initiated and not completed yet. The procedure considers or reconsiders the pair (β_{in},p) and updates sat and

dp accordingly. The core of the procedure is only executed when (β_{in}, p) is not suspended and not in the domain of the dependency graph. In effect, if (β_{in}, p) is suspended, no subcomputation should be initiated and if (β_{in}, p) is in the domain of the dependency graph, it means that none of the elements upon which it is depending have been updated. Otherwise a new computation with (β_{in}, p) is initiated. The core of the procedure is a repeat loop which computes the lower approximation of (β_{in}, p) given the elements of the suspended set. Local convergence is attained when (β_{in}, p) is in the domain of the dependency graph. One iteration of the loop computes each of the clauses defining p and computes the union of the results. If the result produced is greater or not comparable to the current value of (β_{in}, p), then the set of abstract tuples is updated. The dependency graph is also adjusted accordingly by removing all elements which depend (directly or indirectly) on (β_{in}, p). Note that the calls to the clauses are done with an extended suspended set since a subcomputation has been started with (β_{in}, p). Note also that, before executing the clauses, the dependency graph has been updated to include (β_{in}, p) (which is guaranteed not to be in the domain of the dependency graph before that update). (β_{in}, p) can be removed from the domain of the dependency graph during the execution of the loop if a pair which it is depending upon is updated.

Procedure `solve_clause` executes a single clause for an input pair and returns an abstract substitution representing the execution of the clause on that pair. It begins by extending the substitution with the variables of the clause, then executes the body of the clause, and terminates by restricting the substitution to the variables of the head. If a literal is concerned with unification, the operations `AI_VAR` and `AI_FUNC` are used. Otherwise, procedure `solve_call` is called and the result is looked up in `sat`. Moreover, if (β_{in}, p) is in the domain of the dependency graph, it is necessary to add a new dependency. Otherwise, (β_{in}, p) needs to be reconsidered anyway.

```
procedure solve(in β_in, p; out sat, dp)
begin
  sat := ∅;
  dp := ∅;
solve_call(β_in, p, ∅, sat, dp)
end.


procedure solve_call(in β_in, p, suspended; inout sat, dp)
begin
  β_in := WIDEN(β_in, p, suspended);
  if (β_in, p) ∉ (dom(dp) ∪ suspended) then
  begin
    if (β_in, p) ∉ dom(sat) then
      sat := EXTEND(β_in, p, sat);
    repeat
      β_out := ⊥;
      EXT_DP(β_in, p, dp);
      for i := 1 to m with c_1, ..., c_m clauses-of p do
      begin
        solve_clause(β_in, p, c_i, suspended ∪ {(β_in, p)},
                     β_aux, sat, dp);
        β_out := UNION(β_out, β_aux);
      end;
      (sat, modified) := ADJUST(β_in, p, β_out, sat);
      REMOVE_DP(modified, dp);
    until (β_in, p) ∈ dom(dp)
  end
end.
```

```
procedure solve_clause(in β_in, p, c, suspended; out β_out;
                       inout sat, dp)
begin
  β_ext := EXTC(c, β_in);
  for i := 1 to m with b_1, ..., b_m body-of c do
  begin
    β_aux := RESTRG(b_i, β_ext);
    switch (b_i) of
      case x_j = x_k:
        β_int := AI_VAR(β_aux);
      case x_j = f(...):
        β_int := AI_FUNC(β_aux, f);
      case q(...):
        solve_call(β_aux, q, suspended, sat, dp);
        β_int := sat(β_aux, q);
        if (β_in, p) ∈ dom(dp) then
          ADD_DP(β_in, p, β_aux, q, dp);
    end;
    β_ext := EXTG(b_i, β_ext, β_int)
  end;
  β_out := RESTRC(c, β_ext)
end.
```

Fig. 2-3: The generic abstract interpretation top down algorithm.

As explained earlier, an abstract interpretation algorithm may not terminate if the computation is made over an infinite domain. The use of a widening is useful to limit the number of abstract inputs to be considered.

The intuition behind this is that an element cannot be refined infinitely often. Each time a call (β_{in}, p) is encountered, the last element of the form (β'_{in}, p) inserted in the suspended set (which now has to be considered a stack) is searched. If such an element exists, the computation continues with $(\beta'_{in} \cup \beta_{in}, p)$ instead of (β_{in}, p); otherwise, the computation proceeds normally.

This operation is executed at the beginning of the procedure `solve_call`.

2.2.5.6. BOTTOM-UP ALGORITHM

The bottom-up algorithm uses the same underlying operations such as the manipulation of the dependency graph, the abstract operations, etc.

One major difference is that the bottom-up algorithm is divided in two phases. The first one computes the outputs whereas the second one catches the inputs.

In the first phase, when it solves a call to a predicate, it forgets its actual input substitution and replaces it with a substitution where each term is bound to `any`. This is thus a bottom-up approach of the problem. This implies that the `RESTRG` operation is skipped.

The second phase complete the set of abstract tuples with the input substitutions for every predicate encountered during the computation. This second phase is very useful because we are in fact more interested by the inputs than the outputs.

The fundamental principles are explained on the following example. Suppose the Prolog program:

```
p(X1,X2,X3,X4):- ..., q(X2,X4), ...
q(X1,X2):- X1 = X2.
```

The abstract domain used here is able to represent the groundness of a term and binding of terms[13]. We suppose that the abstract substitution while computing `p/4` associates X_2 to `ground` and X_4 to `any`. The abstract substitution resulting from the computation of `q/2` tells us that X_2 and X_4 are both associated to `any` (because of forgetting the input) and are bound together. Now, thanks to the `EXTG` operation performed after the returning from the call to `q/2`, we know that X_4 is `ground` (in effect, as X_2 is `ground` and X_2 and X_4 are bound, we can deduce that X_4 is `ground`).

Notice that in general, the bottom-up algorithm can lose some accuracy with regard to the top-down algorithm. On the other hand, as the number of iterations is often smaller, the bottom-up algorithm is faster.

2.2.5.7. SEQUENCE BASED TOP-DOWN ALGORITHM

When using a sequence of substitutions as an abstract domain for the output, the algorithm gets a little different. The problem is to have accurate sequence of substitutions; that is to say, we do not want to make a simple least upper bound after the interpretation of clause. Instead, we prefer to use the `CONC` operation which performs the concatenation of substitutions. This operation is quite complex but the main idea is to refine the exclusion

[13] This is in fact the PROP abstract domain explained later.

between clauses of a predicate. These exclusions may be due to an executed cut, arithmetic predicates, incompatibility between the input substitutions of the clauses, etc. In fact, it depends on what the abstract domain is able to catch.

For example, let us consider a predicate made of two clauses. If the abstract interpretation of the first clause binds a variable `X` to a predicate `f(...)` while `g(...)` is bound to the second and we know that this predicate will be called with `X/ground`; we can deduce that these two clauses are exclusives because if one of them leads to a result, the other will surely fail.

Here are some more examples. In the following piece of program, the first and the second clause of predicate `p` are obviously exclusives. There is thus no way to compute the result as the least upper bound; it is cleverer to take this fact into account.

```
p(X,Y):- X > Y, ...
p(X,Y):- X < Y, ...
```

In the next example, we notice that, as the cut of the first clause is surely executed, the abstract result of the next clause does not even have to be considered.

```
p:- !, ...
p:- ...
```

2.3. RESULTS OF AN ABSTRACT INTERPRETATION

2.3.1. INTRODUCTION

The algorithms terminate when all the predicates are stables; i.e., no more result can be refined. It is now time to examine these results. They are stored in the set of abstract tuples. The problem is that this set may contains raw materials that are in fact useful for the fixpoint computation but not relevant for the abstract interpretation itself. In effect, our major concern is to know all possible inputs for a given predicate; but sometimes intermediate computations are developed. Let us have a look at the following example:

```
p([H|T],X):- p(T,Y), q(Y).
p([],[]).
q([]).
```

If we start the abstract interpretation of `p(ground,any)` on the Prop domain, we notice[14] that `q(ground)` is computed although it is obviously not called because the result of the predicate `p(ground,any)`, called before `q` in the first clause of `p`, is `(ground,any)`. It is because this last result is the "stable" result, but before we obtained this result, we had a non-stable result for `p`, telling us the `Y` might be `ground`.

That is why, to separate the good from the bad, so to speak, we introduce the notion of foundation (computed by the mean of a post-processing algorithm).

2.3.2. POST-PROCESSING ALGORITHMS ~ THE FOUNDATION

Many abstract interpretation applications need more information than input/output pairs. Additional information can be computed easily by a post-processing step once the fixpoint has been reached. This approach is attractive for various reasons. On the one hand, it allows the abstract semantics and the fixpoint algorithm to be kept as simple as possible[15]. On the other hand, the additional information can be computed easily given the results of the fixpoint algorithm. Most of these post-processing steps are based on variations of the algorithm depicted in Fig. 2-3.

```
procedure collect(in β_in,p,sat)
begin
    collect_call(β_in,p,∅,sat)
end.
```

[14] In the set of abstract tuples.

[15] In particular, the fixpoint algorithm will be computed more efficiently.

```
procedure collect_call(in β_in, p, suspended, sat)
begin
  β_in := WIDEN(β_in, p, suspended);
  if (β_in, p) ∉ suspended then
  begin
    for i := 1 to m with c_1, ..., c_m clauses-of p do
      collect_clause(β_in, p, c_i, suspended ∪ {(β_in, p)}, sat)
  end
end.

procedure collect_clause(in β_in, p, c, suspended, sat)
begin
  β_ext := EXTC(c, β_in);
  for i := 1 to m with b_1, ..., b_m body-of c do
  begin
    β_aux := RESTRG(b_i, β_ext);
    switch (b_i) of
      case x_j = x_k:
        β_int := AI_VAR(β_aux);
      case x_j = f(...):
        β_int := AI_FUNC(β_aux, f);
      case q(...):
        collect_call(β_aux, q, suspended, sat);
    end;
    β_ext := EXTG(b_i, β_ext, β_int)
  end
end.
```

Fig. 2-4: The basic schema for post-processing algorithms.

The post-processing algorithm is closely related to the generic algorithm. The main differences are:

- all the instructions manipulating the dependency graph are removed;
- the `repeat` loop of procedure `sole_call` has been removed and replaced by a single execution of the body of the loop;
- the `RESTRC` operation is removed because `collect_clause` does not need to return a result;
- the set of abstract tuples is not updated.

We used a post-processing algorithm to compute the *foundation*. The foundation is the set of tuples $(\beta_{in}, p, \beta_{out})$ required to answer the query which triggered the computation.

The Fig. 2-5 shows the modified collect_call procedure for foundation algorithm.

```
procedure collect_call(in β_in, p, suspended, sat, foundation)
begin
  β_in := WIDEN(β_in, p, suspended);
  if (β_in, p) ∉ foundation then
  begin
    foundation := foundation ∪ {(β_in, p)};
    for i := 1 to m with c_1, ..., c_m clauses-of p do
      collect_clause(β_in, p, c_i, suspended ∪
      {(β_in, p)}, sat, foundation)
  end
end.
```

Fig. 2-5: The collect_call procedure for foundation computation.

Additionally, the set foundation is initialized with the empty set in collect procedure and then passed to collect_call and finally to collect_clause. The set suspended is replaced by the set foundation in the membership test of (β_{in}, p). In effect, if the pair (β_{in}, p) is already in the foundation, there is no need to further compute this call.

In fact, speaking about implementation, we do not need a set to store the tuples. We just add a Boolean for each tuple which is true iff the tuple is in the foundation.

2.4. SOME GOALS OF PROLOG ABSTRACT INTERPRETATION

Abstract interpretation is not only a theoretical application; it is useful to optimize Prolog programs during the compilation. In effect, the knowledge of some data properties at some program points permits, for example:

- replacing the general unification algorithm (less efficient) with a specialization which takes into account the specificity of the actual situation;
- a better use of the memory;
- etc.

Here are a few examples of possibilities created by the abstract interpretation of Prolog.

2.4.1. SPECIALIZATION OF THE UNIFICATION ALGORITHM

When the Prolog compiler encounters an instruction like:

X_1 = f(X_2, ..., X_n),

where X_1, X_2, ..., X_n are bound respectively to the terms t_1, t_2, ..., t_n, it builds a term f(t_2, ..., t_n) and then applies the general unification algorithm to t_1 and f(t_2, ..., t_n), because in general t_1, t_2, ..., t_n may be any terms. But in practice, these terms are not any and that is why it is possible to generate a more efficient code.

2.4.1.1. MODE ANALYSIS

If the abstract interpretation reveals the following mode associations for the terms of the previous example: X_1/var and X_2/ground, ..., X_n/ground, the compiler can replace the general unification algorithm by the following code:

```
begin
   new(X1,f/n);   {memory allocation for a predicate f/n}
   X1↑[1]:= X2;   {assignment in X1's 1st cell of X2}
   ...
   X1↑[n-1]:= Xn;
end.
```

This code boils down to a simple sequence of assignments and skips several consistency tests like occur checks. The final situation is depicted in Fig. 2-6 which schematizes the memory.

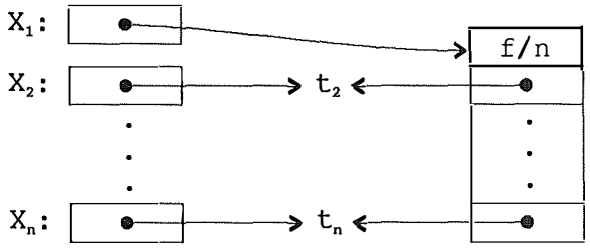


Fig. 2-6: State of the memory after the unification.

Another abstract interpretation could reveal that the following modes are associated: X_1/ground and X_2/var, ..., X_n/var. In that case, the compiler could generate the code:

```
begin
    if struct(X1, f/n)   {tests the consistency of the
                                predicate}
    then begin
              X2:= X1↑[1];
              ...
              Xn:= X1↑[n-1]
    end
end
```

Again, we profit from the same advantages as the one we won in the previous example and the memory situation is the same.

But we forgot to take into account the *sharing*. That is to say if two variables X_i and X_j ($i \neq j$; $i, j \geq 2$) are unified, this unification algorithm leads to an incorrect situation. That is why the sharing analysis is also an important part of Prolog abstract interpretation.

2.4.1.2. SHARING ANALYSIS

As we saw previously, the sharing analysis is an important matter when trying to replace the general unification algorithm by a more efficient one.

We define the relation noshare as:

$$\text{noshare}(X_i, X_j) \Leftrightarrow \text{vars}(t_i) \cap \text{vars}(t_j) = \varnothing,$$

where t_i and t_j are respectively the terms bound to X_i and X_j and vars(t_i) is the set of all the variables present in the structure of the term t_i.

Thus, if we add the following condition (i.e., raised by the abstract interpretation): noshare(X_1, X_2) ∧ ... ∧ noshare(X_1, X_n) then the last specialized unification algorithm is correct.

2.4.1.3. PATTERN ANALYSIS

If we define the relation form as:

form(X) = f/n ⇔ t is formed like f(u_1, ..., u_n),

where t is the term bound to X, and u_i are terms, and if the abstract interpretation of the previous example reports also that form(X_1) = f/n; then we can skip the structure test in the specialized algorithm.

2.4.2. CARDINALITY ANALYSIS

It is sometimes useful to know if a predicate is (strictly) determinist or not.

In the Table 2-2, there are some example of cardinality analysis for the well-known append(X_1, X_2, X_3) procedure:

X_1	X_2	X_3	cardinality	condition
list	list	var	1	noshare(X_1, X_3)∧noshare(X_2, X_3)
ground	ground	var	0 or 1	
var	var	list	at least 1	noshare(X_1, X_2, X_3)[16]

Table 2-2: Cardinality analysis for append(X1, X2, X3).

2.4.3. STATIC REUSE OF THE MEMORY

Memory allocation is a long process for an operating system; that is why it can be useful to detect if some parts of it are never referenced after a certain program point so that it is possible to recycle them.

---

[16] noshare(X_1, X_2, X_3) ⇔ noshare(X_1, X_2) ∧ noshare(X_2, X_3) ∧ noshare(X_1, X_3)

For example, if the abstract interpretation of `append(X1,X2,X3)` reveals that:

- the following modes are associated: `X1/list`, `X2/list` and `X3/var`;
- `noshare(X1,X2,X3)`;
- `X1` is never used afterwards,

then `X3` can point to the same memory zone as `X1` as showed in Fig. 2-7.

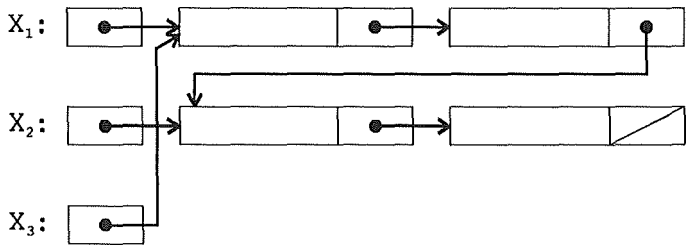


Fig. 2-7: Memory recycling for append(X1,X2,X3).

2.5. REFERENCES

[CHJO95] CHABOT F. and JOUCKEN P., *Abstract Interpretatiion of full prolog*, 1995.

[LCMR] LE CHARLIER B., CORSINI M., MUSUMBU K. and RAUZY A., *Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Symbolic Finite Domains.*

[LECH91] LE CHARLIER B., *L'analyse statique des programmes par interprétation abstraite*, Nouvelles de la Science et des Technologies vol. 9, number 4, 1991.

[LEVA94] LE CHARLIER B. and VAN HENTENRYCK P., *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, ACM Transactions on Programming Languages and Systems (TOPLAS), 1994.

[LECH??] LE CHARLIER B., *Abstract Interpretation and Finite Domain Symbolic Constraints.*

[LVBM94] LE CHARLIER B., BRAEM C., MODART S. and VAN HENTENRYCK P., *Cardinality Analysis of Prolog*, 1994.

[LVM91] LE CHARLIER B., MUSUMBU K. and VAN HENTENRYCK P., *A Generic Abstract Interpretation Algorithm and its Complexity Analysis*, 1991.

3. OBJECT ORIENTED PROGRAMMING AND C++

The purpose of this chapter is to allow a non-Object Oriented programmer or a non-C++ programmer to read this whole report. It delivers the main Object Oriented concepts such as encapsulation, inheritance, polymorphism and dynamic bindings. The last part of this chapter is a brief introduction to the C++ language.

Contents of this chapter:

3.1. INTRODUCTION

The goal of our work was to integrate several applications. These applications all did the same thing (i.e., a Prolog abstract interpretation), but they were using different abstract interpretation options: that is to say different algorithms, different domains, different ways of storing data, etc. Our plan was to build one application that could compute a Prolog abstract interpretation with any combination of the existing options, according to the user choice.

To achieve this, a well-suited technology is the Object Oriented programming. It is an excellent way to represent algorithms and domains as objects because it provides some notable benefits. Encapsulation, inheritance and specialization improve the code cleanness, comprehensibility and reusability. Dynamic bindings allow one to write, for example, generic algorithms that do not need to be changed when we add new domains to the application.

We now review the general notions of Object Oriented programming in order to use these notions freely in this report.

3.2. NOTIONS OF OBJECT ORIENTED PROGRAMMING

3.2.1. CLASS AND OBJECT

We can define an object as a variable (i.e., a region of storage with associated semantics); it is something material (as opposed to a value) which can be created, transformed and destroyed.

As with all variables in typed programming languages, the object proceeds from a type. To be as clear as possible and to distinguish the object and its type, we call a class the object type. An object is an example of a class.

3.2.2. ENCAPSULATION, INHERITANCE AND SPECIALIZATION.

The basic feature of Object Oriented programming is the *encapsulation*. That is to say that all the data needed by an object can be bundled inside. An object can also have its proper procedures, one calls them *methods*. This has essentially a methodological effect; the encapsulation permits a more readable program because it separates the interface from the implementation (as shown on Fig. 3-1). This way, the user is able to use the ob-

ject in a relatively safe and predictable manner without being tempted to peek at the object's implementation. The properties of an object are far more valuable and reusable than the code used to implement them.

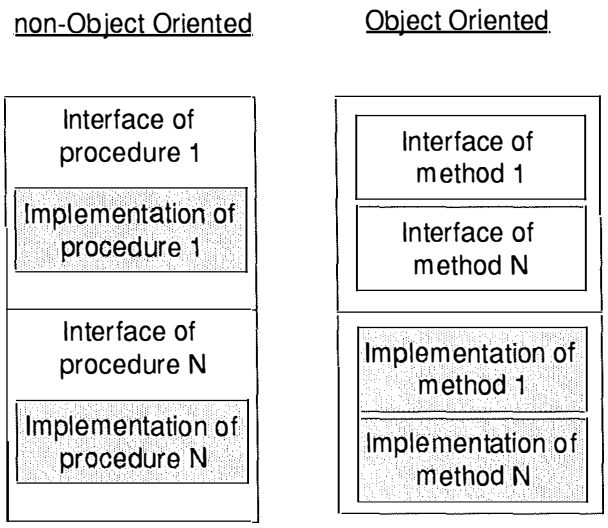


Fig. 3-1: Encapsulation.

It is possible to link two objects with an *inheritance* relationship. The inheritance relationship lets the programmer model the "kind of" relationship between a son and a father object. This notion can be extended to *multiple inheritance*: a son can have several fathers. Note that it does not leave out the fact that a father can have several sons. A son object owns the data and methods defined in its father and inherited by its fathers. Inheritance allows the factorization of the code.

An inherited method can be rewritten in the purpose to adapt it to the new object; this is the notion of *specialization*. Notice that this method always keeps its original signature.

Example:

Let Shape be a class which encapsulates one data and one method:

```
class Shape {
        Data:
                center: point
        Methods:
                Move
},
```

where point is a type for representing spatial position. Let Circle inherit from Shape. Because of that inheritance, the class Circle owns implicitly the data (center) and methods (Move) which are present in the class Shape. Circle is the following class:

```
class Circle inherits from Shape {
        Data:
                radius: integer
        Methods:
                Draw
}.
```

Let Polygon inherit from Shape and be the following object (which contains implicitly the data and methods of Circle):

```
class Polygon inherits from Shape {
        Data:
                number_of_side: integer
        Methods:
                none
}.
```

Let Triangle inherit from Polygon. The object Triangle inherits from the data of Polygon. The method Move is specialized.

```
class Triangle inherits from Polygon {
        Data:
                p1, p2, p3: point
        Methods:
                Move
                Draw
}.
```

Our example's inheritance graph is now as Fig. 3-2:

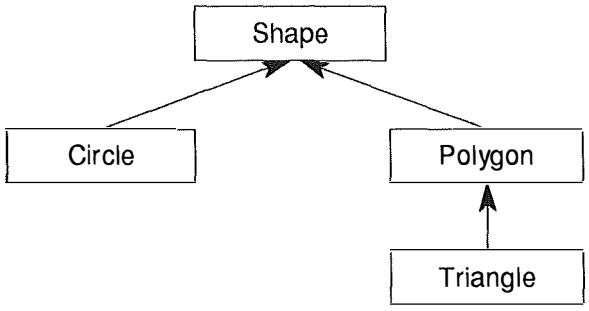


Fig. 3-2: Inheritance graph.

3.2.3. POLYMORPHISM

The inheritance and specialization notions together allow *polymorphism.* A polymorphic method is a method which can be applied to several classes. Notice that a polymorphic method is defined by several distinct methods which have the same names but belong to different classes. In effect, if an object inherits from another and if one of its methods is specialized, this method can be called on both objects (but the method's code is different with regard to the object's type).

For instance, in the previous example, we can apply the method `Move` on an instance of class `Shape`, `Circle`, `Polygon` or `Triangle`. The methods defined in the class `Shape` will be called if we have a `Shape`, `Circle` or `Polygon` object type. The methods defined in the class `Triangle` will be called if we have an object `Triangle` type. The method `Move` is polymorphic.

3.2.4. GENERICITY

3.2.4.1. GENERIC PROCEDURE (DYNAMIC BINDINGS)

A *generic* procedure is a procedure which can be applied to several types. It is different from a polymorhpic procedure because only one code exists and can be executed with different types. For instance in the previous example, a generic procedure could be a procedure that moves and prints any figure. To achieve this, we need two new concepts: the *dynamic bindings* and the *dynamic type.*

Usually, the binding between the method call and the method code is static; it is defined at compilation time. But we can postpone that binding to run time; that kind of method is called *dynamic* (or *virtual*).

The dynamic type is a type that can be, at run time, a specialization of the type at compilation time. A type is static if all its methods (including the inherited methods) are static, otherwise this type is dynamic.

For instance, if we want this procedure[17]

```
procedure Drag&Drop(fig: reference to a Shape)
{
      fig->Move
      fig->Draw
}
```

to be generic, the class `Shape` must be a dynamic type (to allow `fig` to be a reference to a specialization of Shape) and the methods `Move` and `Draw` must be dynamic (to call the right one with regard to the object's type).

Practically, we just need to declare the `Move` and `Draw` methods of the class `Shape` as dynamic in the class `Shape`; thanks to this declaration, the class `Shape` is dynamic. Note that the `Draw` method did not exist in the first version of the class `Shape`. We have to declare it as virtual in the class `Shape` to avoid a compilation error at the second line of the procedure `Drag&Drop`. The problem is that we have no implementation of this method at the `Shape` level. The solution is to declare the `Draw` method as a *pure virtual method* which is a virtual method without implementation. The class `Shape` is now as follows:

```
class Shape {
      Data:
            center: point
      virtual Methods:
            Move
      pure virtual Methods:
            Draw
}.
```

We call a class that has at least one pure virtual method a *pure virtual class*, (example: the classes `Shape` and `Polygon`). As opposed to

---

[17] Where `O->M` represents the application of a method `M` on an object `O`.

this, we call any classes that are not pure virtual classes *concrete classes* (example: the classes `Circle` and `Triangle`).

We can say that a pure virtual method specifies that a method exists on every concrete derived class. It forces concrete derived classes to provide a definition for this method.

Notice that the pure virtual classes cannot be instanciated (be an object), they are only abstractions and they can only be used to manipulate concrete specializations. For instance, in the procedure `Drag&Drop`, `fig` is now a reference to an object which cannot be instanced (`Shape` is pure virtual class). That is why, during the execution, `fig` can only be a specialization of `Shape`.

An advantage of a generic procedure is that it can call a future code. In effect, old polymorphic methods can dynamically bind to new codes. A new derived class can be used by an existing generic procedure without modifying this procedure. For instance, in the previous example, we can add a new derived class from the class `Polygon`: the class `Square`, supposing we implement the inherited pure virtual method Draw (if we do not make this implementation, the class `Square` would be pure virtual!). Without modifying anything to the procedure `Drag&Drop`, it works with a square shape.

3.2.4.2. GENERIC TYPE

The definition of a *generic type* is one which contains one or more undefined types. The goal is to allow the user to create an instance and to choose a type for the undefined type. That notion is very interesting for code reuse. Let us see the following generic class definition:

```
class List of t {
      data:
            head: reference to a type t
            tail: reference to a List of t object
      methods:
            Insert(in data: t)
            GetHead(out data: t)
            GetTail(out next: reference to a List of t
                  object)
}
```

The undefined type of this generic class is `t`. When we create an object `List` we must provide `t`'s type. Thus, we have one code for the object `List` and we can use it to manage lists which store whatever we need.

The use of generic type increases the code reuse and simplifies maintenance because the algorithms are defined once, and then instanciated several times for each type that is needed.

3.3. THE C++ LANGUAGE

3.3.1. C++, A WELL-KNOWN OBJECT ORIENTED LANGUAGE

Every fifteen seconds a non-C++ programmer switches to C++. In October 1991, the number of C++ users was estimated at 400,000. At that time, the C++ community was doubling every seven and a half months. To be ultra-conservative, let us assume that the rate has slowed to doubling every twelve months. Based on these assumptions, on May 2005 every man, woman and child on planet Earth will be a C++ programmer (and the following year we will discover life on Mars, no doubt).

The C++ language supports every general characteristic provided by an Object Oriented language. Encapsulation, inheritance (and multiple inheritance), specialization, polymorphism, generic functions and generic types (which are called "template" in the C++ dialect).

3.3.2. C++, A COMPLEX LANGUAGE

3.3.2.1. INTRODUCTION

The conception of an application is always something difficult to achieve. Even at a high conception level, identifying objects and organizing a correct inheritance graph often takes a lot of time. And it is still more

complicated when we must pay attention to the language we use. In effect, a complex language often contains a lot of complications or restrictions because at times, some programming concepts are incompatible.

In this section, we want to illustrate the difficulty of designing a complex language like C++. As an example, we clarify in this section that multiple inheritance is sometimes incompatible with pointer casting in C++.

3.3.2.2. AN EXAMPLE OF C++ LIMITS

One of the first C++ limitations we encounter is the prohibition of some object pointer casting. This part can be skipped by the reader not interested in technical C++ information. On the other hand, we think it interesting to develop because it causes us some troubles to achieve our goals.

In fact, the C++ compiler does not allow us to cast an object that is issued from a rhombus inheritance. That was our first C++ nightmare. We finally found out the reasons for this incompatibility, which we briefly explain here.

In order to make this example clear, we first review what is a pointer casting; then explain the multiple inheritance and finally reveal that reconciling these two notions is difficult in a programming language like C++.

A. Pointer casting

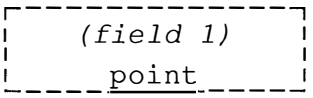
a. Review

Casting a pointer means changing the type of the memory zone it references. Let us have a look at the `Shape` example depicted above:

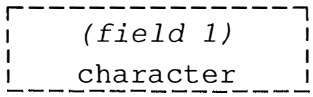
Let us declare `C` a reference to an object `Circle`. It means that the memory that begins at the address referenced by `C` is divided as follows (the first field represents the data inherited from the class `Shape` -the shape's center- and the second represents the data from the class `Circle` -the circle's radius-):

(field 1)	*(field 2)*
point	integer

Now, if we cast (change the pointer's type) `C` to an object `Shape`, the memory (pointed by `C`) is interpreted with the following mask:



As we can see, this memory interpretation (resulting from a casting) seems to be correct because the mask maps the right data in memory (which is the circle's center). In this example, the casting is done correctly. A bad casting example could be the casting of the `C` pointer to a character, because the memory would be interpreted with the following mask:



Here, the data pointed by `C` would be interpreted as a character (and is, in fact, a point). We have what we call a *wild pointer*.

Pointer casting can sometimes be useful. For example, suppose class `Square` is derived from `Shape`, and the method `GetSides` exists on the class `Square` but not on the class `Shape`. As a Square can be manipulated in terms of a Shape (because the class Shape is dynamic), a developer might downcast an object `Shape` reference to an object `Square` in order to access the method `GetSides`.

Notice that the use of pointer casting must be done very carefully because it makes the code difficult to read. Moreover, as shown above, a wrong casting may result in a wild pointer.

b. Upcasting and downcasting with language C++

If a derived class inherits from a base class, C++ lets you convert a reference to a derived class in a reference to a base class because the derived class is a "kind of" the base class; the *upcasting* is implicit. As opposed to this, converting a reference from a base class to a derived class is not allowed in C++. To do this we need to explicitly cast the pointer; it is a *dowcasting*.

B. Multiple inheritance and the C++ virtual base class

As multiple inheritance is allowed, we can imagine the following situation: a class `D` can inherit from the classes `B` and `C` which, in turn, both inherit from the class `A`. We have the following inheritance graph (Fig. 3-3):

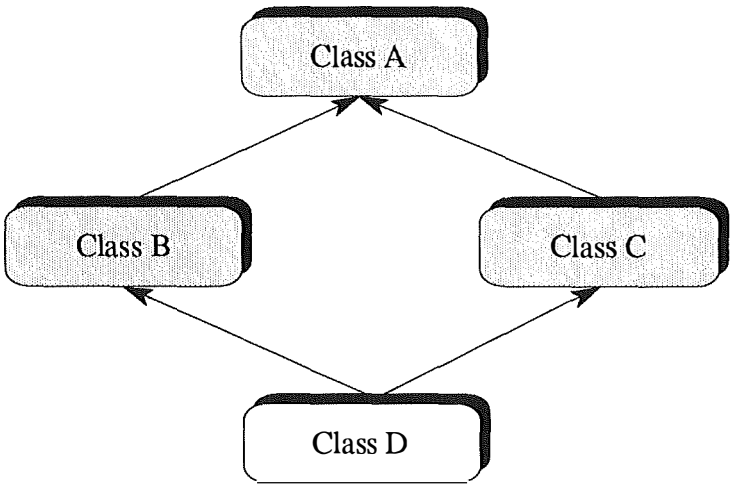


Fig. 3-3: The rhombus inheritance graph.

Classes `B` and `C` do not specialize any methods. The problem is that the class `D` inherits class `A`'s data twice (and needs these data only once). The C++ solution (to have the class `A`'s data only once in the class `D`) is to declare the class `A` as a *virtual base class*. This way, the class `D` is as follows: the data inherited from the class `A` are shared by the methods inherited from classes `B` and `C`.

We can generalize this problem of the rhombus inheritance graph to any class that inherits from two (or more) classes if these classes inherit from the same class.

C. The incompatibility of C++ pointer casting and multiple inheritance

C++ does not allow casting a reference to a virtual base class or a derived class. Here is why.

Let us have a look at the memory allocation for the rhombus inheritance's classes (see Fig. 3-3).

- The memory allocation for the object `A` of a rhombus inheritance:

(field 1)
class A's data

- The memory allocation for the object `B` of a rhombus inheritance:

(field 1)	*(field 2)*
class A's data	class B's data

- The memory allocation for the object C of a rhombus inheritance:

(field 1)	*(field 2)*
class A's data	class C's data

- The memory allocation for the object D of a rhombus inheritance:

(field 1)	*(field 2)*
class A's data	class B's data
(field 3)	*(field 4)*
class C's data	class D's data

Now let us create an object D and initialize the several fields of this object. Imagine that we cast a reference to this object to a pointer to an object C. As explained above, the memory interpretation is done with the object C mask. We now can easily see the problem: the second field in memory represents the class B data (because an object D has been created) and the interpretation of this field is the class C data (because of the new pointer type).

Notice that, with a recent C++ compiler, the problem of casting a virtual base class can be avoid by using a "dynamic casting" which solves, by means of some computation, this problem.

This example was used to show that designing a complex language sometimes leads to some complications or restrictions; here the language must prohibit some classical pointer casting because of the acceptance of a rhombus inheritance graph. A complex language cannot be as didactic as a simple one. It is often more complicated and less intuitive to use.

3.4. REFERENCES

[CLLO95] Marshall P. CLINE and Greg A. LOMOW, *C++ Frequenctly Asked Questions*, Addison-Wesley Publiching Co., 1995.

[LECH1L] LE CHARLIER B., course notes: *Théorie des langages: paradigmes de programmation*, 1° licence.

Part II: Design and implementation

This part mainly focuses on the work we did at Brown University under the supervision of Pascal Van Hentenryck. We explain the features wanted for such a system. We then explain how we built the whole application, component by component and step by step.

4. INTRODUCTION TO THE ABSTRACT INTERPRETER APPLICATION

In this chapter, we lay the foundation of the design of the application. It is an introduction to the global design that is studied thoroughly in the next chapters.

4.1. DESIGNING THE APPLICATION

So now we are ready to start the design of the Prolog abstract interpreter. As shown in the previous sections, there are a lot of properties we want to find out about a Prolog program. Our main objectives concerning the design were:

- Conceiving a lone system able to include all existing features[18] (i.e., different fixpoint algorithms, several domains of abstract substitutions, different organizations of the set of abstract tuples, etc).
- Not only should the system integrate the existing features but it has to be opened enough to be easily extendible. The interfaces between the different components (especially the abstract domains) must suggest a natural way to write add-ons.
- The abstract interpreter project was mature enough to stand a wide distribution. This software tool reveals its power when combined with a Prolog compiler. Again, a convenient interface to the fixpoint algorithms is a key concept to achieve this goal.
- As one gets nothing for free, some concessions in terms of performances should be admitted to fulfill the above objectives. This possible loss of efficiency must be kept into reasonable bounds.
- Without being a real objective but rather a consequence, we hoped the final product would open new kind of results obtained by abstract interpreter units made of components never gathered together.

The core of the application is the fixpoint algorithm. Following the first objective, a perfect way for integration would be to design them as generic procedures. As polymorphism and dynamic bindings allow genericity, our concern is now to find a tool offering these concepts. On the other hand, the interface feasibility would be greatly simplified by data and procedures encapsulation.

The object oriented programming paradigm implements all these concepts and seems thus an appropriate framework to sustain the design of a multi-services provider system.

To achieve the goals of efficiency and wide distribution, we searched in the field of object oriented programming language and realized that C++ was suitable to accomplish this task.

---

[18] A lot of specific abstract interpreters are already implemented.

Being faced for the first time with such a complex assignment, we did not really know where to start. We could have spent our time trying to create a perfect design on paper by considering every component and finding a correct, integrated solution. But then we would not have fulfilled the other part of the task, which was an obligation to get results by means of an executable code.

We first identified the main components to treat as objects:

- the abstract domain,
- the fixpoint algorithm,
- the set of abstract tuples.

So we began to design the interfaces of these various components and pasted the appropriated code behind them. The result was a first version of the application having all the generic features needed for the future but where only one specialization for each of them was implemented.

We then extended this embryonic system with further specialization of the generic components. More than once, we felt that issues were not possible to solve without a complete design reorganization. That was the price to pay for a method close to "trial and error". But, on the other hand, we have a solid result; tested and proven for future extensions.

5. DOMAINS

As one of the most likely extensions of the system is the addition of new domains, their design is an important matter. This chapter retraces the thought process that came before the final design. It also briefly describes the domains already included in the system.

Contents of this chapter:

5.1. ABSTRACT SUBSTITUTIONS AS OBJECTS

As the execution of an abstract interpretation algorithm is a series of actions on abstract substitutions[19] following the Prolog program, we need to have a handy way to manipulate them. That is why it is a good idea to have an object to represent them.

One of the goals of our work was to permit every abstract interpretation algorithm to be executed with any abstract domains. To achieve this, we needed to define a model of abstract substitutions on which the interpretation algorithm can perform the abstract operations. So we came to create a pure virtual object which owns all the abstract operations defined in section I.2.2.4. These methods are the interface that permits modifications or queries of an abstract substitution.

Moreover we can exploit another aspect of object oriented programming (the specialization) to easily extend a domain already implemented. In effect, we can specialize some of its methods in order to create more efficient and/or accurate domains without having to rewrite it from scratch[20].

The consequence on the computational level is that when a specific abstract domain is chosen out of the existing ones, the method call resulting from the application of an abstract operation on a substitution is solved at run-time. We thus lose some efficiency but we win genericity.

5.2. BUILDING THE INHERITANCE GRAPH

The object "abstract substitution" is quite complex. It originates from a long maturing process based on notions such as efficiency, polymorphism and other constraints that are discussed in this section.

We choose to explain its building step by step instead of a straightforward presentation. We begin with a simple model adapted to the abstract substitutions alone. We will soon find that this model is too poor to include the abstract sequence of substitutions notion and that it has to be refined. Finally, it becomes obvious that sometimes the interpretation algorithms

---

[19] We will distinguish abstract substitutions and abstract sequences of substitutions later.

[20] For example Pattern + arithmetic lists is a specialization of Pattern.

need other data than the substitutions when computing and that these data are not part of the substitutions.

5.2.1. THE OBJECT "ABSTRACT SUBSTITUTION"

We define a pure virtual object to represent the abstract substitutions and name it SUBST. Every domain that would be implemented must be depicted by an object that inherits from SUBST and thus respect its interface.

Note that this specialized object can no longer be pure virtual if it has to be used by the abstract interpretation algorithm. But it can sometimes be useful to generate other models (thus pure virtual objects) based on SUBST. It can serve, for example, the following purposes:

- An intermediate model. This kind of model can be used to factorize the code when two (or more) domains are similar. It means that some methods of the object SUBST would be specialized at this level and inherited by these domains.
- A model needed for particular interpretation algorithms. It is possible that some algorithms need more operations or knowledge about a substitution than the ones available in the SUBST model. Such a model restrains this particular algorithm genericity because all domains not inherited from this new model cannot be utilized in this case.

In fact, we never use the last possibility because it is incompatible with our will of genericity of the algorithms. In effect, such a design would restrict the set of combinations of domains and algorithms.

Instead, we define some methods by default at the SUBST level. When it is impossible to avoid it, those kinds of methods throw an error message and abort the computation (which finally is the same as restraining the genericity, as above). But sometimes we can write a method which it returns the less definite result with respect to its knowledge. Of course, it is possible (and even recommended) to specialize this method to gain accuracy.

5.2.2. ADDING THE ABSTRACT SEQUENCES OF SUBSTITUTIONS

It is sometimes useful to have not only a single substitution (computed as the least upper bound of all the results) as an abstract interpretation of a Prolog query but also to have an abstract sequence of all the solutions generated. Including this notion in the abstract interpreter leads us to distinguish input value and output value. We also distinguish the program point value in order to take the cut operation into account.

5.2.2.1. INPUT AND OUTPUT VALUES

Because an abstract sequence of substitutions cannot be used as an input for a query, this new concept forces us to make a difference between the input data and the output data of the interpretation algorithm. We now have two new pure virtual objects which are:

- VALUE_IN: virtual object for representing concrete inputs;
- VALUE_OUT: virtual object for representing concrete outputs.

In effect, it is methodologically correct to distinguish the inputs and the outputs because, in a concrete execution of a Prolog program, an input for a clause is a substitution, while its output is a sequence of substitutions (possibly boiled down to a lone substitution). These two concepts are thus two different objects in our design.

We call SEQ, the abstract sequence of substitutions, a pure virtual object which inherits from the VALUE_OUT object. As an abstract substitution can be both input and output, the object SUBST inherits from both VALUE_IN and VALUE_OUT (see Fig. 5-1).

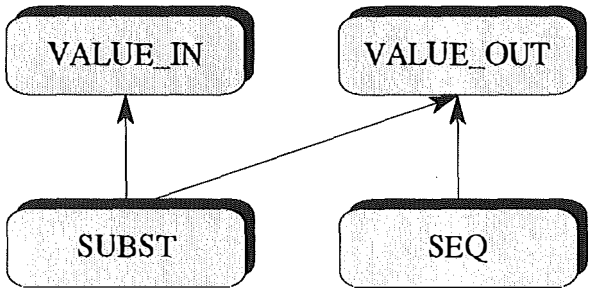


Fig. 5-1: Distinction of VALUE_IN and VALUE_OUT in the inheritance graph of substitutions & sequences.

5.2.2.2. CAPTURING INFORMATION AT PROCEDURE LEVEL

At this stage, the interpretation algorithm manipulates VALUE_OUT objects (that can be either an abstract substitution or an abstract sequence of substitutions) because its behavior is to refine the result of a query for a predicate (that is to say output data) through the iterations. On the other hand, the cut predicate has not "directly" an effect on the structure of output but influences the behavior of the execution. An execution of the cut leads to two effects which are the following (review):

- Effect at the clause level: the execution does not backtrack beyond the cut.
- Effect at the procedure level: the next clauses are not executed.

Thus the cut also influences the abstract interpretation at the procedure level. To capture this effect, it is necessary to hold the cut information from the clause level.

We therefore introduce a third kind of value: VALUE_PRG. This object is manipulated by the algorithms during interpretation in order to hold the information relevant to the computation. Although we currently only need such a value to store the cut information, we choose to create this general object that can be specialized to take into account other information relevant to the procedure level. This object contains a reference to a VALUE_OUT object that represents the abstract (sequence of) substitution in itself.

As a consequence, we immediately specialize VALUE_PRG in VALUE_CUT that contains a Boolean which indicates the encounter of an executed cut.

We have now a new non-virtual object VALUE_PRG that represents a program point value. That object contains a reference to a VALUE_OUT object. That object is specialized into VALUE_CUT to add the cut information to the output value manipulated during the computation. The collection of data that are in use can now be schematized as follows (Fig. 5-2):

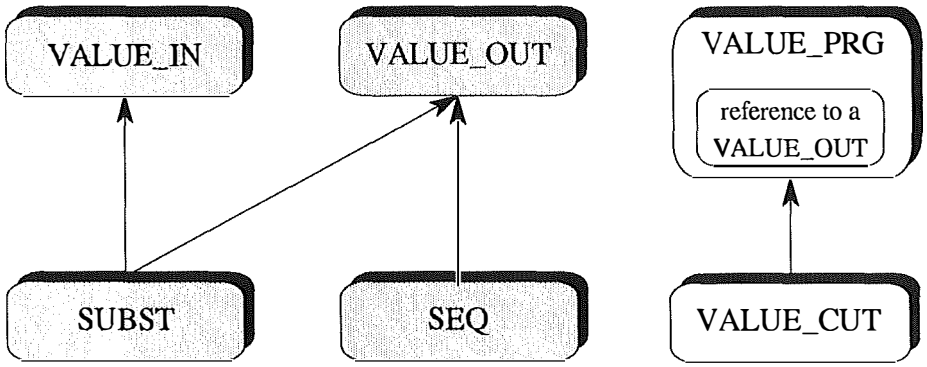


Fig. 5-2: Insertion of the objects VALUE_PRG and VALUE_CUT in the data structure.

5.3. CONCLUSIONS AND IMPLICATIONS

In this section, we leave the virtual part to enter into the real world; that is to say objects which are implementations of abstract domains. Moreover, we discuss the addition of new domains.

5.3.1. DOMAINS ALREADY IMPLEMENTED

We depict in Fig. 5-3 the final result of our work. The PROP, PATTERN, PATTERN + Arithmetic Lists, TYPE_GRAPH, CARTESIAN PRODUCT non-virtual objects inherit from the SUBST virtual object. The CARDINAL non-virtual object inherits from the SEQ virtual object. These abstractions of concrete domains are explained in a few words below.

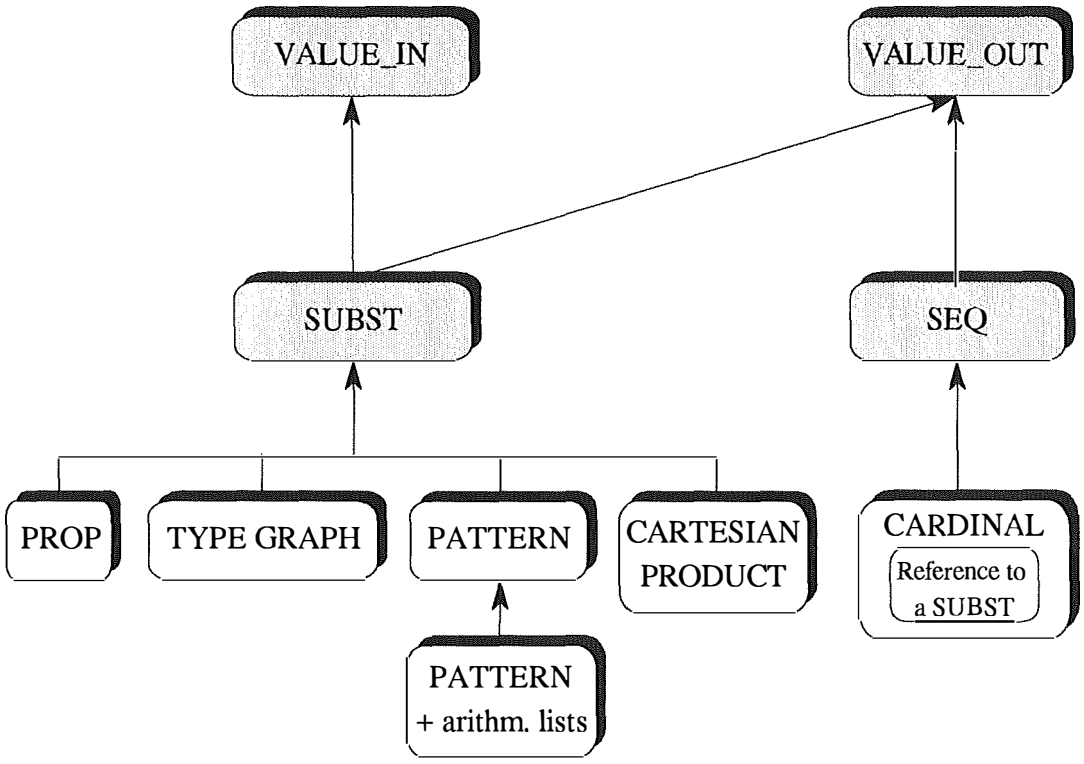


Fig. 5-3: Hierarchy of the implemented domains.

In this section, we have a closer look at each implemented domain. We first explain in a few words the properties they are able to represent; then we illustrate them with an abstract interpretation of the predicate append with the basic top-down polyvariant algorithm.

5.3.1.1. THE DOMAIN PROP

The abstract domain Prop gives information about the groundness of a variable. The concrete substitution over $D = \{X_1, \ldots, X_n\}$ is abstracted by a Boolean formula using variables from D. Every variable is assigned a truth value to denote its groundness (true means ground and false, not ground). This Boolean formula is also built with logical connectives (and, or) and ordered by implication. For example $X_1 \Leftrightarrow X_2$ abstracts the substi-

tutions $\{X_1/Y_1, X_2/Y_1\}$, $\{X_1/a, X_2/a\}$, but not $\{X_1/a, X_2/Y\}$ nor $\{X_1/Y_1, X_2/Y_2\}$.

The following sat is the result of the abstract interpretation of `append(any, any, any)`. The number of goal iterations was three while the number of clause iterations was six.

```
Predicate: append/3
-------------------
- Couple #1:
   Bin:
    ( any any any ) --> 1
   Bout:
    ( any any any ) --> (~X3 & (~X2) + (X2 & ~X1))
                        + (X3 & X2 & X1)
```

We can concretize the resulting output abstract substitution with, for example:

- $\{X_1/[], X_2/Y, X_3/Z\}$ instanciation of `(~X3 & ~X2)`;
- $\{X_1/Y, X_2/[a], X_3/Z\}$ instanciation of `(~X3 & X2 & ~X1)`;
- $\{X_1/[a], X_2/[b], X_3/[c]\}$ instanciation of `(X3 & X2 & X1)`.

But not with:

- $\{X_1/Y, X_2/[b], X_3/[c]\}$.

5.3.1.2. THE DOMAIN TYPE-GRAPH

Although Prolog is an untyped programming language, type analysis is important since it allows us to specialize the general unification algorithm, for example.

The domain Type-graph permits us to infer in a term its disjunctive and recursive structures by the means of subterms. Its grammar can be described, using the BNF rules, as:

$$\langle T\rangle ::= \langle constant\rangle \mid OR(\langle T\rangle_1, \langle T\rangle_2) \mid p(\langle T\rangle_1, \ldots, \langle T\rangle_n) \mid @(\langle T\rangle) \mid any$$

where:

- $\langle constant\rangle \in$ Constant, the set of the constants,
- $OR(\langle T\rangle_1, \langle T\rangle_2)$ represents the disjunction between $\langle T\rangle_1$ and $\langle T\rangle_2$,
- $p(\langle T\rangle_1, \ldots, \langle T\rangle_n)$ indicates that the subterm is instanciated by a predicate p/n containing the subterms $\langle T\rangle_1$, ..., $\langle T\rangle_n$,

- `@(<T>)` indicates that the term `<T>` is recursively present in the subterm,
- `any` represents any term.

The following sat is the result of the abstract interpretation of `append(any, any, any)`. The number of goal iterations was four while the number of clause iterations was eight.

```
Predicate: append/3
-------------------
- Couple #1:
   Bin:
    ([1],[2],[3]) -->
     graph:
      [1] 1:Any
      [2] 1:Any
      [3] 1:Any
   Bout:
    ([1],[2],[3]) -->
     graph:
      [1] 1:OR (
                  2:.(
                        3:Any ,
                        @(1)
                     )
                  4:[]
                )
      [2] 1:Any
      [3] 1:Any
```

We notice that the structure of the subterm `[1]` is recursive and is, in any case, a list while the two other subterms `[2]` and `[3]` are `any`.

We can concretize the resulting output abstract substitution with, for example:

- $\{X_1/[], X_2/Y, X_3/Z\}$;
- $\{X_1/[a|[X|[]]], X_2/Y, X_3/Z\}$.

5.3.1.3. THE DOMAIN CARDINAL SEQUENCE

The idea of the abstract sequence of substitutions is to catch all the successive results of a Prolog query. The goal here is to capture the number of solutions (minimum and maximum) and the termination (sure termination, sure non-termination and possible termination) of the execution. It also in-

cludes an object SUBST which represents the substitution like in the previous domains.

The following sat is the result of the abstract interpretation of `append(any, any, any)` computed with abstract sequence of substitutions Prop. The number of goal iterations was four while the number of clause iterations was eight.

```
Predicate: append/3
-------------------
- Couple #1:
   Bin:
    ( any any any ) --> 1
   Bout:
    [0,oo]  possible termination
    ( any any any ) --> (~X3 & (~X2) + (X2 & ~X1)) + (X3
                        & X2 & X1)
```

With regard to the interpretation with substitution, the interpretation with sequence adds the following information: the Prolog program can produce an infinity of solutions and may terminate.

5.3.1.4. THE DOMAIN PATTERN

The key concept of the abstract domain Pattern is the notion of subterm. Given a substitution on a set of variables, an abstract substitution associates the following information with each subterm appearing in the substitution:

- its mode,
- its pattern which specifies the main functor as well as the subterms which are its arguments,
- its possible sharing with other subterms.

Each subterm is identified unambiguously by indices. For instance, the substitution

```
{X1/[a,b], X2/[c,d], X3/Ls}
```

could have for instance the following indices association:

```
{(1,[a,b]), (2,a), (3,b), (4,[c,d]), (5,c), (6,d),
                    (7,Ls)}.
```

In this domain, the mode is taken from the following set M = {⊥, ground, var, ngv, novar, gv, noground, any}[21]. This set satisfies the ordering depicted in the Hasse diagram at Fig. 5-4. An oriented vertex from one node to another denotes that the first is greater than the second.

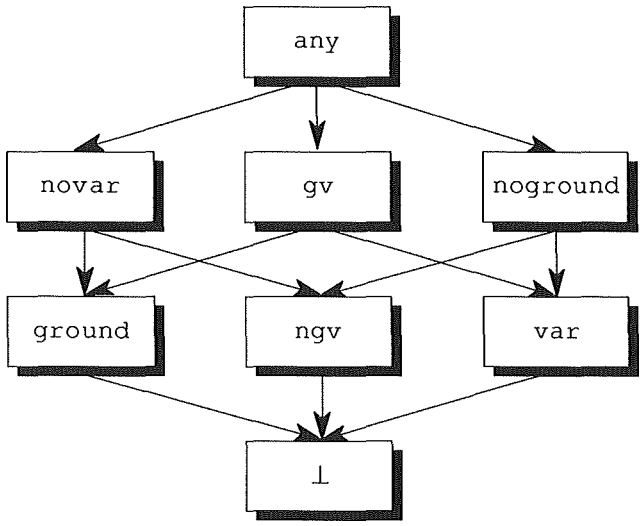


Fig. 5-4: The ordering of the modes.

The mode association for the example substitution would be:

{(1,ground), (2,ground), (3,ground), (4,ground), (5,ground), (6,ground), (7,var)}.

The pattern component possibly[22] assigns to an indice an expression $f(i_1, \ldots, i_n)$ where f is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. In our example, the pattern component makes the following associations:

{(1,.(2,3)), (2,a), (3,b), (4,.(5,6)), (5,c),(6,d)}[23].

Finally, the sharing component specifies which indices, not associated with a pattern, may possibly share variables. Attention is restricted to indices without patterns since the patterns already express some sharing information and we do not want to introduce inconsistencies between the

---

[21] Ngv stands for "no ground no variable", gv for "ground or variable" and any is the top element.

[22] In fact, the pattern is optional.

[23] .(head,tail) is the usual list constructor.

components. In our example, the only sharing is the couple `(7,7)`, expressing that variable `Ls` shares a variable with itself.

In order to clarify the concept, a more appealing representation is given for the predicate `append/3` instanced with the above substitution:

```
append(ground(1):.(ground(2):a, ground(3):b),
ground(4):.(ground(5):c, ground(6):d), var(7))
```

with the sharing information `{(7,7)}`.

The following sat is the result of the abstract interpretation of `append(var, var, var)`. The number of goal iterations was four while the number of clause iterations was eight.

```
Predicate: append/3
-------------------
- Couple #1:
   Bin:
    (Var(1),Var(2),Var(3))
    ps: {1,1} {2,2} {3,3}
   Bout:
    (Novar(1),Any(2),Noground(3))
    ps: {1,1} {1,2} {1,3} {2,1} {2,2} {2,3} {3,1} {3,2}
{3,3}
```

We can concretize this resulting output abstract substitution with, for example:

- $\{X_1/[],X_2/Y,X_3/Z\}$;
- $\{X_1/[a|X],X_2/Y,X_3/[a|Z]\}$.

5.3.1.5. THE DOMAIN PATTERN + ARITHMETIC LISTS

The domain Pattern + arithmetic lists is a specialization of the domain Pattern. It inherits from this last and thus owns all its features. However, it can hold more information about the terms of a substitution in the form of an arithmetic list. As a consequence, some of the methods inherited from the domain Pattern are specialized to take this information into account[24].

[24] The implementation of the Pattern + arithmetic list domain was facilitated by the object oriented programming.

Prolog permits the insertion of arithmetic built-ins like $<$, $\leq$, $>$, $\geq$, $\neq$ and $=$. They can be used to ensure some data properties within a clause because if they fail, the execution of the current clause stops.

Adding information about arithmetic built-ins is only useful when computing abstract interpretation over a domain of abstract sequences. In effect, arithmetic lists are used to detect exclusive clauses.

In the following example, the domain Pattern + arithmetic lists is able to catch the fact that the two clauses are exclusives.

```
p(X):- X < 0, ...
p(X):- X ≥ 0, ...
```

This example is quite obvious but this domain is more clever and is able to catch the same information in the following modified program which has the same semantic.

```
p(X):- lt(X,0), ...
p(X):- get(X,0), ...

lt(X,Y):- X < Y.
get(X):- X ≥ Y.
```

Moreover, this domain will compute the transitive closure of the arithmetic conditions in order to avoid missing information. In effect, if we have the two arithmetic relations `X > Y` and `Y > Z`, we can deduce `X > Z`.

To illustrate this domain, we do not use the classical `append/3` predicate because the results are the same as the domain Pattern. Instead, we want to point out the gain of accuracy given by the arithmetic lists in cardinality analysis. We thus first compute an abstract interpretation of the program `partition/4` with an abstract sequence using the domain Pattern and then compare the results with the addition of arithmetic lists.

The predicate `partition/4`, given a number `X`, splits a list `L` into two lists L_1 and L_2 where all the elements of L_1 (respectively L_2) are smaller or equal (respectively greater) to `X`. The text of `partition/4` is:

```
partition([],X,[],[]).
partition([F|T],X,[F|S],B):-
    F <= X,
    partition(T,X,S,B).
partition([F|T],X,S,[F|B]):-
    F > X,
    partition(T,X,S,B).
```

The result of the abstract interpretation of `partition(ground, ground, var, var)` with a sequence algorithm based on Pattern tells us, in four goal and twelve clause iterations, that

- the output substitution is `(ground, ground, ground, ground)`,
- the query may terminate,
- the number of solution is between `0` and $+\infty$.

On the other hand, adding the arithmetic lists to the domain Pattern leads to the following conclusions in three goal and nine clause iterations:

- the output substitution is `(ground, ground, ground, ground)`,
- the query may terminate,
- the number of solution is between `0` and `1`.

To sum up, this last domain informs us that `partition(ground, ground, var, var)` is deterministic. Moreover, the result is reached in less iterations than with the domain Pattern.

5.3.1.6. THE CARTESIAN PRODUCT OF DOMAINS

The Cartesian product of domains is what we (almost) got for free[25] with object oriented programming. It is a Cartesian product of two domains (both can be Cartesian product also). The algorithm executes "in parallel" both components.

Every domain present in a Cartesian product does not communicate with another, except if a substitution turns to bottom then they all become bottom. That is why sometimes it is possible to gain accuracy with a Cartesian product compared to a domain executed alone. In effect, if a domain catches the bottom and forwards it to the other component then, when a least upper bound is performed (at the end of each clause) on this last, the result could be less general.

---

[25] It was coded in a few lines.

For example: suppose that we use only the domain Prop and the interpretation of the first clause returns `(ground, var, any)` and the second returns `(var, var, var)`, then the least upper bound is `(any, var, any)`. In a second interpretation, we use a Cartesian product of the Prop and domains Pattern. Let us say that the result for the first clause is different from bottom and the second is bottom for the domain Pattern. So Pattern communicates this bottom result to Prop. Then, on the Prop side, the least upper bound would be between `(ground, var, any)` and $\perp$, and will lead to `(ground, var, any)` which is more precise than previously.

It is possible to have another type of product where the communication would be more intense[26]. A domain could ask, for example, if a variable is ground to another one and consequently act.

5.3.2. THE ADDITION OF NEW DOMAINS

In order to add a new domain, one must of course implement a non-virtual object which inherits from SUBST (or SEQ) and specialize the pure virtual methods. As told earlier it is also possible to start from an already implemented domain and to specialize it.

5.4. REFERENCES

[BRMO94] BRAEM C. and MODARD S., *Abstract Interpretation for Prolog with cut: Cardinality Analysis*, 1994.

[CHJO95] CHABOT F. and JOUCKEN P., *Abstract Interpretatiion of full prolog*, 1995.

[LVBM94] LE CHARLIER B., BRAEM C., MODART S. and VAN HENTENRYCK P., *Cardinality Analysis of Prolog*, 1994.

[LVC94] LE CHARLIER B., CORTESI A. and VAN HENTENRYCK P., *Evaluation of the domain Prop*, 1994.

[LVC294] LE CHARLIER B., CORTESI A. and VAN HENTENRYCK P., *Type Analysis of Prolog using Type Graphs*, 1994.

[PVH96] VAN HENTENRYCK P., personal communication.

[26] See 10.1.1.

6. ALGORITHMS

This chapter describes the design and implementation of the algorithms. As the algorithms manipulate several generic types that have not been explained yet, we first discuss the implementation of these concepts, that is to say the management of the set of abstract tuples and the Prolog program code. To finish this chapter, we expose the advantage of a post-processing treatment.

Contents of this chapter:

6.1. DATA STORAGE ~ SET OF ABSTRACT TUPLES

6.1.1. INTRODUCTION

The results of an abstract interpretation computation are incrementally built. We must choose a way to store these results gradually. The management of this storage has already been introduced in the theoretical part of this report: the organization of the set of abstract tuples.

The representation of this set of abstract tuples is an important choice in the implementation. There are different ways to implement the two main operations that are necessary: operations EXTEND and ADJUST.

Remember that a *set of abstract tuples* (sat) is a set $(\beta_{in}, p, \beta_{out})$ where p is a predicate of arity n and β_{in}, β_{out} are abstract substitution on variables $X_1, \ldots, X_n$. β_{in} represents the input substitution for the predicate p and β_{out} the output substitution.

To be more efficient, we distinguish the *global* sat and the *local* sat. A local sat for a predicate p is a set of couples of input and output substitution $(\beta_{in}, \beta_{out})$ for this predicate. We call the global sat (for a Prolog program P), the set of local sat $\{ls_1, \ldots, ls_m\}$ where m is the number of predicates declared in the program P; ls_i is a local sat of a particular P's predicate (we have only one local sat per predicate recorded in the program P).

Note that in our application, the global sat is implemented by means of a hash table pointing to the local sat's.

6.1.2. SAT IN MONOVARIANT AND POLYVARIANT ALGORITHMS.

When performing an abstract interpretation, a decision must be made about keeping track of the $(\beta_{in}, \beta_{out})$ couples of local sets of abstract tuples. In a local sat, we can either store a couple for each different β_{in} or store only one couple $(\beta_{in}, \beta_{out})$ where β_{in} is the union of the input substitutions for the predicate of the local sat. We use these sat in a *polyvariant* and a *monovariant* algorithm respectively.

6.1.2.1. SAT IN A POLYVARIANT ALGORITHM

As told above, a sat used in a polyvariant algorithm contains several tuples (one for each different β_{in} of the local sat's predicate). There are

different ways of managing that set of tuples. Our application contains a Hasse diagram implementation and a hash table implementation.

A. Hasse diagram

A local sat for a predicate `p` can be represented by a Hasse diagram (see section I.2.1.3.2). Each abstract tuple (relevant to `p`) encountered by the algorithm is a node of the diagram. If `a` and `b` are respectively the nodes $(\alpha_{in}, \alpha_{out})$ and $(\beta_{in}, \beta_{out})$, then `a` $\leq$ `b` iff $\alpha_{in} \leq \beta_{in}$.

With this kind of implementation, the two main sat operations (`EXTEND` and `ADJUST`) are implemented as:

- `EXTEND`$(\beta$, `local_sat`$)$: this operation looks up in the `local_sat`'s Hasse diagram if a couple (β, β_{out}) exists. If it doesn't, extend returns a `local_sat'` which is extended (with regard to the `local_sat`) with a new couple $(\beta, \perp)$. (Notice that instead of $\perp$ as the output substitution, we could access the tuple's descendants to compute an initial approximation).
- `ADJUST`$(\beta, \beta'$, `local_sat`$)$: this operation find the couple (β, β_{out}) in the `local_sat` and returns a `local_sat'` which is updated with this new result. More precisely, the value of `sat'`$(\beta")$ for all $\beta" \geq \beta$ is equal to `lub`$(\beta'$, `sat`$(\beta"))$ and all other values are left unchanged.

Let us illustrate the way that kind of sat is managed. The abstract domain used in this example is able to find out if a term is `ground`, `var` or `any`. Imagine we look in the local sat of a predicate `p/3`. Suppose we have already interpreted `p/3` with the following input substitutions (it means also that we extended the sat with them): `(any, any, ground)`, `(any, ground, var)`, `(any, ground, ground)`, `(ground, any, ground)`, `(ground, ground, ground)`. The Hasse diagram local sat would look like Fig. 6-1 (Notice that on that figure, only the input substitution of a node are shown, not the output substitution).

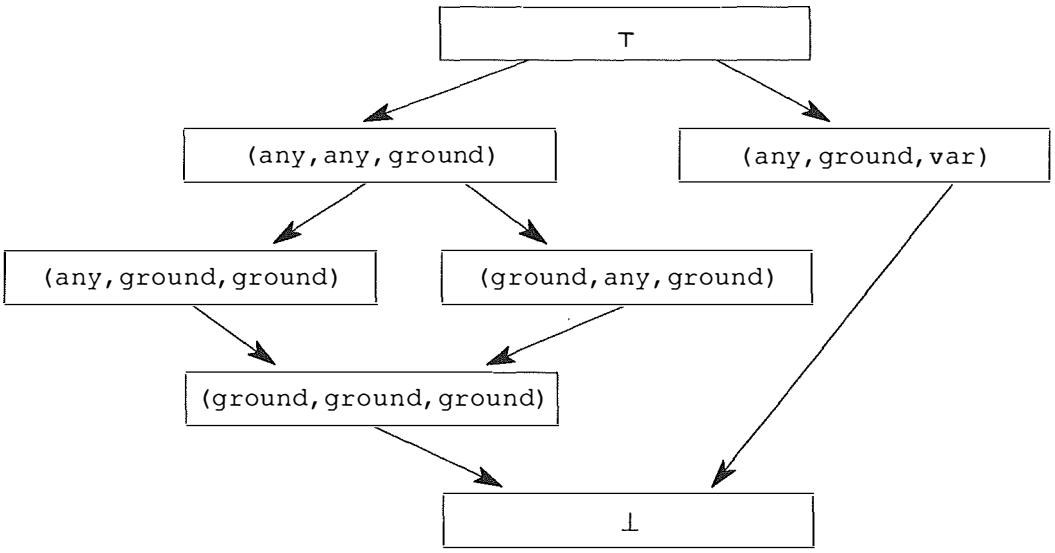


Fig. 6-1: Example of a Hasse diagram local sat.

Now, if we execute the ADJUST operation on the node (ground, ground, ground) with β_{out}, the following points would happen:

- the output substitution of the node (ground, ground, ground), would be updated with β_{out};
- all the nodes in which its input substitution is more general (>) than (ground, ground, ground) would be updated with the result of the least upper bound of the node's old output substitution and β_{out}. That is to say that the following nodes would be updated: ($\top$), (any, any, ground), (any, ground, ground) and (ground, any, ground). In fact, more precisely, ADJUST updates the output substitution of a node and then starts a depth-first search algorithm to update its ancestors. Each time a node is updated, each father is considered for possible updating.

B. Hash table

The hash table implementation simply stores each couple (β_{in}, β_{out}) in a hash table. The β_{in} is the key of that hash table and must then be hashable.

Here, the implementations of the two main operations are:

- EXTEND(β, local_sat): this operation looks up in the local_sat's hash table if a couple (β, β_{out}) exists. If it does not, it returns a local_sat' which is extended (with regard to the local_sat) with a new couple (β, $\bot$). Note that it is quicker to extend the hash table be-

cause there is no order; here we simply must hash the input substitution, and add the new node.

- `ADJUST(β,β',local_sat)`: this operation finds the couple (β, β_{out}) in the `local_sat` and returns a `local_sat'` which is updated with this new result. More precisely, the value of `sat'(β)` is equal to $lub(\beta', \beta_{out})$ and all other values are left unchanged. This `adjust` operation is quicker than the one implemented for the Hasse diagram because here we must simply modify one node (not its ancestors).

As the hash table type is widely used in our application, notice the hash table is designed as a generic type (a template in the C++ jargon).

6.1.2.2. MONOVARIANT SAT

Instead of storing several abstract couples in the local sat, we can store only one couple $(\beta_{in}, \beta_{out})$, where β_{in} is the union of the input substitutions for the predicate of the local sat.

The main operations are implemented the following way:

- `EXTEND(β,local_sat)`: this operation looks up the couple $(\beta_{in}, \beta_{out})$ of the `local_sat` and returns a `local_sat'`. The couple of the `local_sat'` is: $(lub(\beta, \beta_{in}), \beta_{out})$.
- `ADJUST(β',local_sat)`: this operation looks up the couple $(\beta_{in}, \beta_{out})$ of the `local_sat` and returns a `local_sat'`. The couple of the `local_sat'` is: $(\beta_{in}, lub(\beta', \beta_{out}))$.

6.1.2.3. COMPARISON OF THE DIFFERENT SAT IMPLEMENTATION

A. Polyvariant versus monovariant

Using a sat used in a monovariant or a polyvariant algorithm may, in general, lead to different results. With the first one, the β_{out} substitution for a given β_{in} can be less accurate than with the second one because of the "least upper bound" in the `ADJUST` operation. Moreover, using sat running with a monovariant algorithm sometimes leads to fower algorithm iterations than a polyvariant sat.

Let us show these differences with an illustration. Consider the following Prolog program (but don't try to catch the meaning of this program!):

```
q(A,B):-
      p(A),
      p(B).

p(X).
```

Consider that the above Prolog program is interpreted with the following options: a top-down algorithm, a substitution output, the domain Prop and an input substitution: `(any,ground)`. We show that interpreting `q` with a polyvariant algorithm (using a hash table sat) leads to three goal iterations and with a monovariant algorithm leads to only two goal iterations.

As we can see on the trace, an interpretation of `q` with a polyvariant algorithm (Fig. 6-2) is as follows. The first literal to call during the computation of `q` is `p(any)` -line 04-. The result of this call is `any` -line 11-. At this moment, the local sat of `p` contains one tuple `{((any),(any))}`. Then the second literal `p(ground)` is computed. The result of this call is `ground` -line 20-. Now, the local sat of `p` contains two tuples `{((any),(any)), ((ground),(ground))}`.

```
(01) Call PRO-GOAL q: ( any ground ) --> X2
(02)    Try clause 1
(03)    Exit EXTC: ( any ground ) --> X2
(04)    Call PRO-GOAL p: ( any ) --> 1
(05)       Try clause 1
(06)       Exit EXTC: ( any ) --> 1
(07)       Exit RESTRC: ( any ) --> 1
(08)       Exit clause 1
(09)       Exit LUB: ( any ) --> 1
(10)       Adjust
(11)    Exit PRO-GOAL p: ( any ) --> 1
(12)    Exit EXTG: ( any ground ) --> X2
(13)    Call PRO-GOAL p: ( ground ) --> X1
(14)       Try clause 1
(15)       Exit EXTC: ( ground ) --> X1
(16)       Exit RESTRC: ( ground ) --> X1
(17)       Exit clause 1
(18)       Exit LUB: ( ground ) --> X1
(19)       Adjust
(20)    Exit PRO-GOAL p: ( ground ) --> X1
(21)    Exit EXTG: ( any ground ) --> X2
(22)    Exit RESTRC: ( any ground ) --> X2
(23)    Exit clause 1
(24)    Exit LUB: ( any ground ) --> X2
```

```
(25)    Adjust
(26) Exit PRO-GOAL q: ( any ground ) --> X2
```

Fig. 6-2: Trace with a polyvariant algorithm.

Let us now use a monovariant algorithm (Fig. 6-3). After the first call to `p` -line 04-, the local sat contains `{((any),(any))}`. Then we have to call for the second time `p` -line 13- with the substitution `(ground)`. The difference with the polyvariant algorithm is here. Remember that only one couple can be inserted in a sat of a monovariant algorithm and that to `EXTEND` the local sat we just take the least upper bound of the inputs. In our example, the new input is `(ground)` and `ground < any` (and so `lub(ground,any) = any`). Because `((any),p) ∈ dom(dp)`, no computation is done and the result of `p(any)` is `sat((any),p)` which is `(any)`.

```
(01) Call PRO-GOAL q: ( any ground ) --> X2
(02)    Try clause 1
(03)    Exit EXTC: ( any ground ) --> X2
(04)    Call PRO-GOAL p: ( any ) --> 1
(05)       Try clause 1
(06)       Exit EXTC: ( any ) --> 1
(07)       Exit RESTRC: ( any ) --> 1
(08)       Exit clause 1
(09)       Exit LUB: ( any ) --> 1
(10)       Adjust
(11)    Exit PRO-GOAL p: ( any ) --> 1
(12)    Exit EXTG: ( any ground ) --> X2
(13)    Call PRO-GOAL p: ( ground ) --> X1
(14)    Exit PRO-GOAL p: ( any ) --> 1
(15)    Exit EXTG: ( any ground ) --> X2
(16)    Exit RESTRC: ( any ground ) --> X2
(17)    Exit clause 1
(18)    Exit LUB: ( any ground ) --> X2
(19)    Adjust
(20) Exit PRO-GOAL q: ( any ground ) --> X2
```

Fig. 6-3: Trace with a monovariant algorithm.

We notice that the result of `p(ground)` is different according to the sat used. With the monovariant algorithm, the result is less accurate. In this case, the result of the computation (`q(any,ground)`) is the same thanks to the `EXTG` operation.

For example, we would lose some accuracy if we were trying to interpret the same predicate with the Pattern domain and with the input substitution `(ground,var)`. Using the polyvariant algorithm, the result would be `(ground,var)`. It is different with the monovariant algorithm; the result would be `(ground,gv)`. Let us think about this interpretation. At the time of the first call to `p` (i.e., `p(ground)`), the sat would be `{((ground),(ground))}`. At the second call to `p` (i.e., `p(var)`), the sat would become `(lub(ground,var), lub(ground,var))` which is equal `((gv),(gv))`. So `sat((ground),p) = (gv)`. Notice that because `sat((any),p)` has been modified, `((ground,var),q)` $\notin$ `dom(dp)` and so needs to be reconsidered; in this case, this reconsideration does not change any of the results.

To conclude about the comparison of polyvariant and monovariant algorithms, we notice that in general, the monovariant algorithm allows the interpretation computation to be quicker. On the other hand, some results can be less accurate with this algorithm.

B. Hasse diagram versus hash table

We can also sometimes see some differences between the Hasse diagram and the hash table implementation.

The Hasse diagram implementation stores the tuples in such a way that the sat operations `EXTEND` and `ADJUST` take more time than in the hash table implementation (because of the order and of the ancestors updating). On the other hand the Hasse diagram implementation sometimes leads to fower goal iterations. In effect, it is possible that due to the Hasse diagram organization, the output substitution for a tuple does not need to be computed because it has been updated by ADJUST operations on some of its sons' nodes.

6.2. MANAGEMENT OF THE PROLOG CODE

6.2.1. BASIC CODE MANIPULATION

Once the Prolog program text is parsed, it is compiled and stored in an object named CODE. Because the algorithms need to peel the Prolog text from the program to the literal, this object can be queried.

For instance, let us take a Prolog program in which the `append/3` procedure[27] appears. First, we can ask to CODE object for that predicate. We receive an object 1 that contains the append procedure:

```
Obj1:   append(X1,X2,X3):-
              X1 = [],
              X2 = X3.
        append(X1,X2,X3):-
              X1 = [X4|X5],
              X3 = [X4|X6],
              append(X5,X2,X6).
```

Now, we can query this object 1 to obtain the first clause of this procedure, and we get an object 2 containing

```
Obj2:   append(X1,X2,X3):-
              X1 = [],
              X2 = X3.
```

We can ask, for instance, the object 1 the clause following the object 2 and we get

```
Obj3:   append(X1,X2,X3):-
              X1 = [X4|X5],
              X3 = [X4|X6],
              append(X5,X2,X6).
```

We can ask the object 3 the number of temporary variables in the clause (it returns three in this case) or the first literal and we get an object 4:

```
Obj4:   X1 = [X4|X5].
```

We can ask the literal's opcode; here it is a unification. We can ask the object 3 for the literal following the object 4. And if we ask it again, we have the last literal:

```
Obj5:   append(X5,X2,X6).
```

Here, the operating code is a call to a goal and thus we can ask more to object 5: the name of the called predicate, the list of arguments, etc. In fact, we can obtain from that kind of object everything relevant for its specific interpretation; it is of course based on its operating code.

---

[27] That append procedure is normalized.

6.2.2. PREFIXING THE CLAUSES

Another way to query a clause object is to consider it formed from three different parts (instead of a list of literal): a prefix, a predicate call and a suffix. Each of them could be possibly empty.

When we parse a clause, we receive a prefix which is the longest list of built-ins, the first predicate call, and the suffix. The parsing operation can be repeated on the prefix. In other words, when considering the prefixes of clauses, it comes to the same thing as peeling in reverse; i.e., beginning with the last literal of a clause and heading towards the first.

For example, if we parse the object 3, we receive the three following objects:

```
Obj6:   X1 = [X4|X5],
        X3 = [X4|X6],

Obj7:   append(X5,X2,X6).

Obj8:   <empty>.
```

Where the object 6 is the prefix, 7 is the predicate call and 8 is the suffix.

This way to consider clauses leads us to a more efficient algorithm rightly nicknamed "prefix". We now only explain the topics relevant to the clauses; the rest of this algorithm is explained later.

Let us consider the following piece of Prolog program[28]:

⓪ $p(\ldots) :- l_1, l_2.$ ⊙

where l_1 and l_2 are built-ins. For a given input abstract substitution ⓪, the output abstract substitution ⊙ is stable; i.e., never has to be computed again. If we add to this program a call to a goal g_1 as last literal of the clause, we obtain the program

⓪ $p(\ldots) :- l_1, l_2,$ ① $g_1.$ ⊙

we notice that the only way for the output substitution ⊙, given an input substitution ⓪, to be modified is if the result of g_1 is refined for the input substitution ①. Thus, if the result is refined, all we have to do to (maybe)

---

[28] The circled numbers like ⓪, ①, ②, ... represent the substitution for this program point.

refine p is to take the result of the computation of g_1. We now modify this program again and add the literal l_3 and l_4 that are not predicate calls:

⓪ p(...):- l_1, l_2,① g_1,② l_3, l_4.⊙

Again, if the output substitution ② of the predicate g_1, given the input substitution ①,is modified, all we have to do when reconsidering the predicate p is to compute the result of the sequence of literal l_3, l_4 for the input ②.

In short, if we modify an algorithm to take these observations into account (i.e., adding on the one hand a set of abstract tuples where predicates are replaced by prefixes, and on the other hand dependencies between goals and prefixes), we can gain significant computation time by skipping unnecessary computations.

6.3. ALGORITHMS AS OBJECTS

At first, the advantage of an object oriented algorithm seems obvious for the factorization of the code and because the paradigm of object oriented programming helps us to express the idea of an algorithm easier. All the data concerning the computation are hidden in the encapsulation and the inheritance property helps us to implement several algorithms that are similar.

Moreover, the algorithms themselves need to be manipulated by the main program, which is independent from the kind of algorithm used. That is why the algorithms need to be generic types in such a way that we can apply a virtual method that asks to solve a query.

6.4. ALGORITHMS ALREADY IMPLEMENTED

6.4.1. INTRODUCTION

As explained earlier, our system wants to be suitable for a large number of ways to tackle abstract interpretation. Several domains fulfill this objective, but several algorithms complete it. In this section, we describe the algorithms that are part of our application.

6.4.2. ALGORITHMS HIERARCHY

We depict the algorithms objects hierarchy in Fig. 6-4. The object named SOLVE is the top pure virtual object that represents the abstract interpretation algorithms. The TOP-DOWN, BOTTOM-UP and SEQUENCE (top down) objects inherit from SOLVE and are pure virtual. For each of

them, we have four algorithms which are the Cartesian products of {prefix, no prefix} and {monovariant, polyvariant}. Thus, we have sixteen non-virtual objects which represent the algorithms.

The algorithm named SEQUENCE is a top-down algorithm that returns sequences of substitutions. We distinguish the algorithms that return substitutions and those that return sequences; these two kind of algorithms are slightly different.

This structure was built gradually. We first laid the foundation for the basic top down polyvariant non-prefix algorithm, and then expanded that foundation little by little to come to this final inheritance graph.

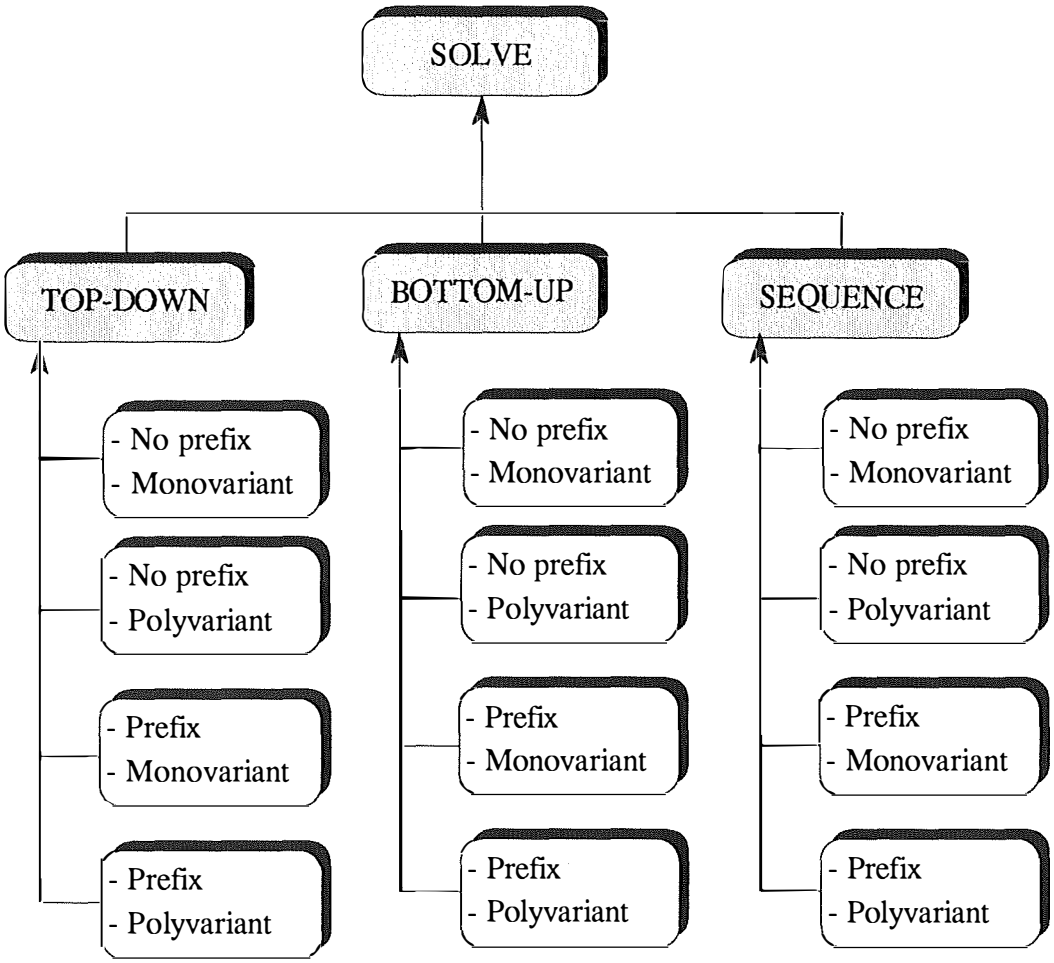


Fig. 6-4: Inheritance graph of the algorithms.

In general, we can observe that the bottom-up algorithms sometimes lose accuracy with regard to the top-down algorithms as predicted in the theoretical part of this report. On the other hand the benchmarks show that

these bottom-up algorithms are often quicker than the top-down algorithms. For general comparisons see the evaluation chapter III.9. and for a full algorithm trace see the section III.7.4.

6.5. REFERENCES

[LEVA94] LE CHARLIER B. and VAN HENTENRYCK P., *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, ACM Transactions on Programming Languages and Systems (TOPLAS), 1994.

[PVH96] VAN HENTENRYCK P., oral communication.

Part III: Utilization and evaluation

In this part, we expose the results of our work. That is to say how a "final user" can consider the application. We explain how to handle the application (with a basic text interface) and how a programmer can integrate it in a compiler. We also evaluate the efficiency of several parameters within a series of benchmarks. We then outline some possible future extensions.

7. HANDLING THE APPLICATION

This chapter depicts the existing interface between our application and a user. The last part of this chapter describes the full trace of a Prolog abstract interpretation.

Contents of this chapter:

7.1. INTRODUCTION

Remember that the object of our work was to integrate several existing applications. The goal of these existing applications were to abstractly interpret a Prolog program; but each application had some particular features like a particular algorithm, domain or data storage. Our plan was to build one system that could compute a Prolog abstract interpretation with any combination of the existing features, according to the user's choice.

More precisely, before thinking about the conception of the new application, we noticed the following existing features for an abstract interpretation (reminds).

- The output of the interpretation computation can be either a abstract substitution or a abstract sequence of substitutions.
- An interpretation can be computed with a top-down or a bottom-up algorithm. Moreover, either a prefix or no-prefix algorithm version can be used.
- The algorithm can keep track of the set of abstract tuples in two different ways: It can be monovariant or polyvariant. Moreover, the polyvariant algorithm data can be stored in a Hasse diagram or a hash table.
- Various domains can be used to compute an abstract interpretation: the Prop domain, the Pattern domain, the Pattern and arithmetic list domain and the Type Graph domain.

Our application is able to compute an abstract interpretation with any combination of these options.

7.2. WHAT THE USER MUST SPECIFY TO COMPUTE AN ABSTRACT INTERPRETATION

Because we wanted the new application to support each of the features depicted above, to compute an abstract interpretation the user has to specify the features to be used.

We had to keep in mind that the new application could be extended in the future and that this application was designed to be incorporated in a Prolog compiler. That is why we allowed the possibility for adding other compilers to our application.

Finally, we summarize the application options as follows:

- the compiler type,

- the algorithm type (top down with a substitution output, bottom up with a substitution output, top down with a sequence of substitutions output),
- a prefixed algorithm or not,
- the algorithm data storage type (monovariant or polyvariant),
- if using a polyvariant algorithm, a Hasse diagram or a hash table implementation can be used,
- the domain (Prop, Pattern, Pattern+la, Type-graph or Cartesian product of two other domains).

A dialog between the application and the user could be:

```
Abstract interpretation options:
Enter the file name ?
> append.p
Enter the predicate name ?
> append
Enter the arity ?
> 3
Enter the mode nb. 1 ?
> ground
Enter the mode nb. 2 ?
> ground
Enter the mode nb. 3 ?
> var
Enter the compiler type
 (1) builtin
> 1
Enter the algorithm type
 (1) top down (abstract substitution output)
 (2) bottom up (abstract substitution output)
 (3) top down (abstract sequence output)
> 1
Prefix ?
 (1) no prefix
 (2) prefix
> 2
Enter the kind of sat ?
 (1) monovariant
 (2) polyvariant
> 2
```

```
Enter the implementation of SAT
 (1) hasse_diagram
 (2) hash table
> 2
Enter the implementation of substitutions
 (1) prop
 (2) pattern
 (3) pattern+list arithm
 (4) type_graph
 (5) cartesian_product
> 2
```

7.3. HOW THE USER CAN QUERY THE RESULTING SAT

The result of a Prolog query is the output substitution. But in order to detect the properties inherent in the input Prolog program after an abstract interpretation, it can be useful to examine the set of abstract tuples. That is why when the abstract interpretation is computed by the application, the user can query the resulting set of abstract tuples.

The next queries are available after the execution:

- get the inputs for a predicate;
- get the input least upper bound for a predicate;
- get the outputs for a predicate;
- get output least upper bound for a predicate;
- print all the set of abstract tuples;
- compute the foundation.

7.4. A FULL EXAMPLE

We introduce a full example of an abstract interpretation. Fig. 7-1 is the trace of the interpretation upon the append Prolog program. The following interpretation options were chosen:

- the top-down algorithm,
- a no-prefix version of the algorithm,
- a polyvariant algorithm with a hash table to implement the set of abstract tuples,
- the Prop substitution domain,
- the query is: `append(ground,ground,any).`

The first iteration of the top-down algorithm is shown at lines 01 to 23. The first clause extends the input substitution (it has no effect since all variables appear in the head) -line 03-. It then binds the first argument with `[]` -line 04- and the second argument with the third -line 06-. Then the substitution is restricted to the head variables (this has again no effect in this case). The union of the previous result with the new one is done on line 10: `lub(⊥,(ground, ground, ground))`. The algorithm then computes the second clause with the input substitution. It extends the input substitution, performs some unifications and calls itself recursively with the same substitution -line 17-. As the algorithm is currently computing `append(ground, ground, any)`, no computation can be initiated for that predicate; that is why the output substitution (i.e., ⊥) is simply picked up in the set of abstract tuples. The second clause then returns ⊥. After, the union -line 22- of the results, the output substitution is `(ground,ground,ground)` and the set of abstract tuples is updated -line 23-.

A second iteration is necessary since the predicate depends on itself and `sat((ground, ground, ground), append)` has been modified (with `(ground, ground, ground)`). This second iteration does not provide more accurate result. The interesting point is at the time of the recursive. The look-up in the set of abstract tuples now returns `(ground, ground, ground)` -line 40- and then, it produces the result `(ground, ground, ground)` -line 42-. Notice that this new result is the same as the one already stored in the set of abstract tuples (computed at the first iteration).

```
(01) Call PRO-GOAL append: ( ground ground any ) --> X2 &
                                X1
(02)   Try clause 1 -iteration #1-
(03)    Exit EXTC: ( ground ground any ) --> X2 & X1
(04)    Call UNIFFUNC: ( ground ground any ) --> X2 & X1
(05)    Exit UNIFFUNC: ( ground ground any ) --> X2 & X1
(06)    Call UNIFVAR: ( ground ground any ) --> X2 & X1
(07)    Exit UNIFVAR: ( ground ground ground ) --> X3 &
                          X2 & X1
(08)    Exit RESTRC: ( ground ground ground ) --> X3 & X2
                          & X1
(09)   Exit clause 1 -iteration #1-
(10)   Exit LUB: ( ground ground ground ) --> X3 & X2 &
                       X1
(11)   Try clause 2 -iteration #1-
```

```
(12)   Exit EXTC: ( ground ground any any any any ) -->
                    X2 & X1
(13)   Call UNIFFUNC: ( ground ground any any any any )
                      --> X2 & X1
(14)   Exit UNIFFUNC: ( ground ground any ground ground
                      any ) --> X5 & X4 & X2 & X1
(15)   Call UNIFFUNC: ( ground ground any ground ground
                      any ) --> X5 & X4 & X2 & X1
(16)   Exit UNIFFUNC: ( ground ground any ground ground
                      any ) --> (~X6 & X5 & X4 & ~X3
                      & X2 & X1) + (X6 & X5 & X4 & X3
                      & X2 & X1)
(17)   Call PRO-GOAL append: ( ground ground any ) -->
                           X2 & X1
(18)   Exit PRO-GOAL append: bottom
(19)   Exit EXTG: bottom
(20)   Exit RESTRC: bottom
(21)  Exit clause 2 -iteration #1-
(22)  Exit LUB: ( ground ground ground ) --> X3 & X2 &
                 X1
(23)  Adjust
(24)  Try clause 1 -iteration #2-
(25)   Exit EXTC: ( ground ground any ) --> X2 & X1
(26)   Call UNIFFUNC: ( ground ground any ) --> X2 & X1
(27)   Exit UNIFFUNC: ( ground ground any ) --> X2 & X1
(28)   Call UNIFVAR: ( ground ground any ) --> X2 & X1
(29)   Exit UNIFVAR: ( ground ground ground ) --> X3 &
                     X2 & X1
(30)   Exit RESTRC: ( ground ground ground ) --> X3 & X2
                    & X1
(31)  Exit clause 1 -iteration #2-
(32)  Exit LUB: ( ground ground ground ) --> X3 & X2 &
                 X1
(33)  Try clause 2 -iteration #2-
(34)   Exit EXTC: ( ground ground any any any any ) -->
                   X2 & X1
(35)   Call UNIFFUNC: ( ground ground any any any any )
                      --> X2 & X1
(36)   Exit UNIFFUNC: ( ground ground any ground ground
                      any ) --> X5 & X4 & X2 & X1
(37)   Call UNIFFUNC: ( ground ground any ground ground
                      any ) --> X5 & X4 & X2 & X1
```

```
(38)   Exit UNIFFUNC: ( ground ground any ground ground
                        any ) --> (~X6 & X5 & X4 & ~X3
                        & X2 & X1) + (X6 & X5 & X4 & X3
                        & X2 & X1)
(39)   Call PRO-GOAL append: ( ground ground any ) -->
                                X2 & X1
(40)   Exit PRO-GOAL append: ( ground ground ground ) --
                                > X3 & X2 & X1
(41)   Exit EXTG: ( ground ground ground ground ground
                    ground ) --> X6 & X5 & X4 & X3 & X2
                    & X1
(42)   Exit RESTRC: ( ground ground ground ) --> X3 & X2
                      & X1
(43)  Exit clause 2 -iteration #2-
(44)  Exit LUB: ( ground ground ground ) --> X3 & X2 &
                 X1
(45)  Adjust
(46) Exit PRO-GOAL append: ( ground ground ground ) -->
                            X3 & X2 & X1
```

Fig. 7-1: Trace of `append(ground, ground, any)`.

8. INTEGRATION IN A PROLOG COMPILER

An abstract interpretation algorithm can be fun as a standalone program but this application may be more useful when integrated into a Prolog compiler. That is why, when designing the application, we always kept in mind the interconnection feature.

This chapter describes a possible design where the abstract interpreter is seen as a server.

8.1. INTEGRATION IN A PROLOG COMPILER

The current compiled version of the application is sufficient to itself. But the main procedure (i.e., the control of different parts of the program: compilation of the program, fixpoint algorithm and results exploitation) could be easily rewritten to take the interfacing with a compiler into account.

We now expose a possible design for a collaboration between a Prolog compiler and the abstract interpreter. The solution would be to consider our application as a server[29] for the compiler. The compiler would ask the interpreter to compute an abstract interpretation with some parameters (the Prolog program code and the interpretation options). Then the compiler could query the interpreter to have some particular details of the resulting set of abstract tuples.

Remember that the input Prolog program code must be parsed before either the compilation or the interpretation. Instead of doing that work twice, we choose, in this design, the Prolog Compiler to do these parsing operations. As a consequence, the interpreter must use the resulting parsed data. To allow our interpreter to read the parsing Prolog program, a solution is to make the Prolog code data of the compiler be an object and inherit from the pure virtual methods of our virtual class CODE. This way, the Prolog compiler specializes these methods and our interpreter is able to query the object that contains the Prolog program code. The methods that must be specialized are simply the ones needed to peel the Prolog text from the program to the literal.

An example of this partnership is depicted in Fig. 8-1.

[29] Referring to the Client/Server relationship.

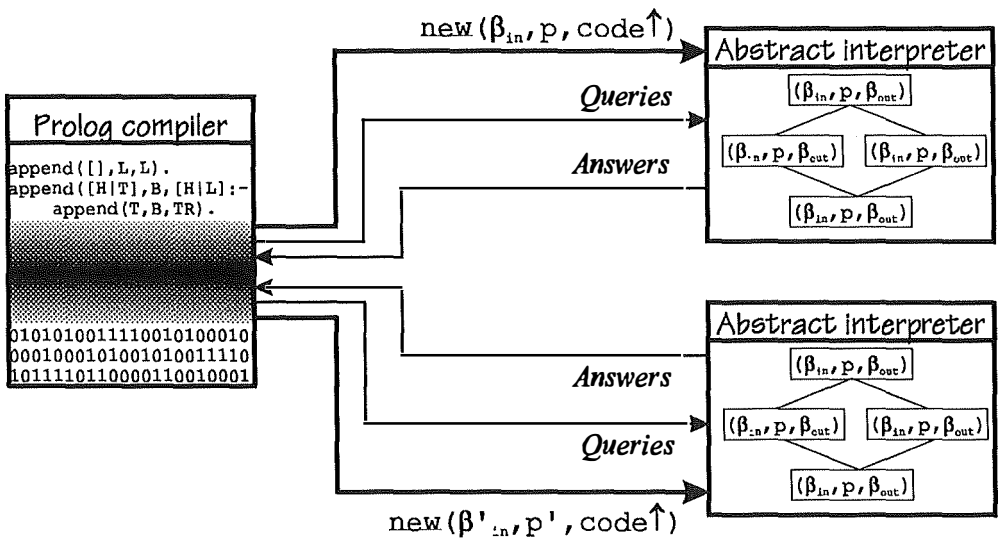


Fig. 8-1: Example of design.

With such a partnership, the following operations can be made by the two components:

- first, the compiler parses the input Prolog program, and obtains an object (`code`) that is readable by the interpreter;
- then, the compiler can ask an abstract interpretation of a predicate (`p`) of the input Prolog program. The compiler must provide the object `code`, the predicate `p` to compute, the input substitution and the interpretation options;
- the interpreter computes `p` according to the object `code`.
- if the compiler needs some details of the computation (i.e., inputs for a predicate, outputs for a predicate, etc), it can ask the interpreter, which then consults the set of abstract tuples to answer.

9. EXPERIMENTAL RESULTS

This chapter contains general comparisons between several interpretations coupled with particular options. The Prolog code of the benchmarks and their goals are described in the annex.

Contents of this chapter:

The tables of this chapter depict the cpu time needed when using particular options of the abstract interpretation. As a consequence of the algorithms' genericity, almost every combination of options is possible. We can use any domains combined with any algorithms. The benchmarks naturally show that using an improper combination of options often leads to huge and unnecessary cpu time.

9.1. DOMAIN PROP

Prefix	No	No	Yes	Yes
SAT	Monovariant	Polyvariant	Monovariant	Polyvariant
peephole	2.28	1.71	2.15	1.90
read	2.24	2.94	2.81	3.14
press	4.32	6.34	4.18	6.20
kalah	1.23	1.26	1.42	1.42

Table 9-1: Cpu time (sec) for top-down algorithm with Prop.

The following table depicts an improper abstract interpretation. In effect, using abstract sequence with the Prop domain is not interesting because the Prop domain is not able to catch the exclusivity of clauses. The algorithm thus spends unnecessary time trying to catch information that the Prop domain cannot manage.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	11.19	6.79	11.02	7.10
read	5.30	6.05	5.38	6.28
press	6.46	10.72	6.24	10.02
kalah	1.66	1.70	1.75	1.78

Table 9-2: Cpu time (sec) for top down algorithm with Prop sequences.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	3.09	3.27	2.85	2.91
read	2.43	3.40	2.54	3.57
press	2.63	5.02	2.46	4.95
kalah	1.68	1.92	1.72	1.93

Table 9-3: Cpu time (sec) for bottom up algorithm with Prop.

9.2. DOMAIN PATTERN

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	3.88	3.75	3.18	4.00
read	4.39	14.57	3.45	13.90
press	4.39	18.53	4.19	16.00
kalah	3.39	4.41	2.70	3.37

Table 9-4: Cpu time (sec) for top-down algorithm with Pattern.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	11.71	11.59	11.12	11.75
read	10.47	30.77	9.92	31.41
press	8.22	30.68	8.03	27.97
kalah	4.61	5.74	3.88	5.17

Table 9-5: Cpu time (sec) for top down algorithm with Pattern sequences.

Prefix	No	No	Yes	Yes
SAT	Monovariant	Polyvariant	Monovariant	Polyvariant
peephole	2.72	4.11	2.65	4.15
read	3.70	12.49	3.53	12.58
press	2.82	7.03	2.68	6.62
kalah	2.86	8.02	2.39	7.52

Table 9-6: Cpu time (sec) for bottom up algorithm with Pattern

9.3. DOMAIN PATTERN + ARITHMETIC LISTS

Prefix	No	No	Yes	Yes
SAT	Monovariant	Polyvariant	Monovariant	Polyvariant
peephole	4.22	4.14	3.53	4.47
read	4.37	15.40	3.99	14.94
press	4.92	19.61	4.74	17.37
kalah	3.47	4.55	2.87	3.83

Table 9-7: Cpu time (sec) for top-down algorithm with Pattern+la.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	14.18	13.77	13.50	13.65
read	9.74	28.87	9.21	28.48
press	8.96	32.83	8.69	30.17
kalah	5.69	5.84	4.55	5.32

Table 9-8: Cpu time (sec) for top down algorithm with Pattern+la sequences.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	2.90	4.43	2.87	4.34
read	4.02	13.17	3.91	13.29
press	3.02	7.58	2.94	7.50
kalah	3.01	8.36	2.51	7.79

Table 9-9: Cpu time (sec) for bottom up algorithm with Pattern+la.

9.4. DOMAIN TYPE-GRAPH

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	4.09	3.12	4.01	3.81
read	/	/	/	/
press	/	/	/	/
kalah	0.08	0.06	0.12	0.08

Table 9-10: Cpu time (sec) for top-down algorithm with Type-graph.

The following table depicts an improper abstract interpretation. As the Type-graph domain is not able to catch the exclusivity of clauses, using abstract sequences with the Type-graph domain is stupid. The algorithm spends unnecessary time trying to catch information that the Type-graph domain cannot manage.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	17494.5	17640.4	17882.2	21335.7
read	/	/	/	/
press	/	/	/	/
kalah	0.13	0.15	0.16	0.25

Table 9-11: Cpu time (sec) for top down algorithm with Type-graph sequences.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	2.02	2.45	2.29	2.89
read	/	/	/	/
press	/	/	/	/
kalah	0.15	0.16	0.19	0.21

Table 9-12: Cpu time (sec) for bottom up algorithm with Type-graph.

9.5. DOMAIN CARTESIAN PRODUCT

The Cartesian product is not efficient since the components do not communicate[30]. In effect, the following tables often show that a computation for a Cartesian product of A and B takes more time than the sum of the times to compute A and B. This phenomenon is explained by the fact that during the computation, the two domains are waiting for each other instead of warning.

9.5.1. CARTESIAN PRODUCT OF PROP AND PATTERN

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	7.96	8.06	6.87	8.37
read	8.28	34.12	7.66	32.95
press	10.44	46.15	9.87	40.28
kalah	5.15	6.86	4.72	6.42

Table 9-13: Cpu time (sec) for top-down algorithm with Cp(Prop, Pattern).

[30] See section 10.1.1. for an open product of domains.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	70.94	63.48	68.45	63.74
read	35.48	85.61	34.93	81.85
press	18.02	70.42	17.63	62.40
kalah	6.78	8.74	6.09	8.34

Table 9-14: Cpu time (sec) for top down algorithm with Cp(Prop, Pattern) sequences.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	7.10	8.77	6.40	8.21
read	10.70	27.62	10.44	27.36
press	9.18	19.69	8.55	19.03
kalah	7.68	12.66	7.16	11.83

Table 9-15: Cpu time (sec) for bottom up algorithm with Cp(Prop, Pattern).

9.5.2. CARTESIAN PRODUCT OF PATTERN AND TYPE-GRAPH

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	10.61	8.60	9.21	9.40
read	/	/	/	/
press	/	/	/	/
kalah	0.12	0.08	0.15	0.21

Table 9-16: Cpu time (sec) for top-down algorithm with Cp(Pattern, Type-graph).

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	35.95	24.74	34.36	24.95
read	/	/	/	/
press	/	/	/	/
kalah	0,10	0.14	0.22	0.20

Table 9-17: Cpu time (sec) for top down algorithm with Cp(Pattern, Type-graph) sequences.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	6.38	7.67	6.29	7.50
read	/	/	/	/
press	/	/	/	/
kalah	0.28	0.32	0.37	0.43

Table 9-18: Cpu time (sec) for bottom up algorithm with Cp(Pattern, Type-graph).

9.5.3. CARTESIAN PRODUCT OF PROP, PATTERN AND TYPE-GRAPH

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	17.09	14.01	14.19	14.44
read	/	/	/	/
press	/	/	/	/
kalah	0.14	0.18	0.17	0.18

Table 9-19: Cpu time (sec) for top-down algorithm with Cp(Prop, Cp(Pattern,Type-graph)).

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	127.00	76.53	121.67	76.47
read	/	/	/	/
press	/	/	/	/
kalah	0.19	0.15	0.34	0.29

Table 9-20: Cpu time (sec) for top down algorithm with Cp(Prop, Cp(Pattern,Type-graph)) sequences.

Prefix SAT	No Monovariant	No Polyvariant	Yes Monovariant	Yes Polyvariant
peephole	10.72	12.74	9.57	11.79
read	/	/	/	/
press	/	/	/	/
kalah	0.51	0.50	0.54	0.63

Table 9-21: Cpu time (sec) for bottom up algorithm with Cp(Prop, Cp(Pattern,Type-graph)).

10. CONCLUSIONS AND FUTURE WORKS

As the main goal of our work was to lay the foundation of an extendible system, there are still a lot of possible enlargements and optimizations that could be added to the system. This chapter doesn't pretend to quote every possible extension but only provides some of them.

Contents of this chapter:

10.1. CONCLUSION

Overall, we laid the foundation for an extendible abstract interpreter. We fulfilled the assignment: namely, improving genericity of a Prolog abstract interpreter through object oriented design. We built several objects to model the different parts that we were able to identify as components of an abstract interpretation system. When we began to work on this project, we had lots of disparate materials (i.e., several versions of abstract interpreters) and when we had to stop we left a new framework able to be extended in the future.

In effect, there is still a lot of material which could be integrated within the framework. For instance, one might want to add new algorithms in order to catch other information. On the other hand, it is also possible to insert other implementations of the set of abstract tuples in order to store the information differently or to increase efficiency.

Obviously, a major interest could be the insertion of other domains. Some of them are already implemented in other applications (and thus have to be adapted to the actual matrix) while others are still in the world of thoughts and still to be built. The fantasy would be to have an application where one can choose from a huge list of domains. Every domain having its specificity and its strength, it is now possible to fine-tune the abstract interpretation of a given Prolog program and a particular goal. Moreover, as we are in a world of communication, why not let the domains communicate with each other. We thus introduce the notion of open product.

10.1.1. OPEN PRODUCT DOMAIN

The idea behind the open product domain is the combination of several domains; it is thus close to the Cartesian product domain. The major difference is that while the components of a Cartesian product stupidly perform the abstract interpretation in parallel, the open product allows them to communicate to speed up the execution time and refine the result.

A domain able to take advantage of the open product feature, would query other components of the open product in order to collect information not in its possession (e.g., groundness of a term, sharing information,...).

Notice that the domains queried must be able to answer; i.e., they must own new methods to collect that kind of information. In general, if one does not want to change the domain to offer this feature, it is possible

to define these methods by default returning less precise information (e.g., "I do not know").

10.1.2. REFINING INPUT SUBSTITUTIONS

In the current version of the abstract interpreter, one can only enter an abstract substitution made of modes although a lot of domains can handle more information (pattern, sharing, etc). A more convenient input interface should help the user to specify more complex queries.

10.1.3. CACHING THE OPERATIONS

In order to decrease the computation time lost in abstract operations, another extension to the system could be to cache them.

Although substitutions are implemented in terms of instanciation of objects (i.e., variables), it is possible to consider them as values[31]. From that notion, we can introduce the abstract operations caching. The idea is to store the result of a given abstract operation on a given abstract substitution so that if the same operation occurs afterwards, simply looking in a table is sufficient to get the result.

Obviously, to implement the caching of abstract operations, none of them could be destructive[32]. Note that this implementation already exists in other frameworks.

---

[31] i.e., an abstract and immaterial value like a number, a set, etc.

[32] This is due to the persistence of a value.

References

[BY95/9] BYTE, A Brief History of Programming Languages, September 1995.

[BRMO94] BRAEM C. and MODARD S., *Abstract Interpretation for Prolog with cut: Cardinality Analysis*, 1994.

[CHJO95] CHABOT F. and JOUCKEN P., *Abstract Interpretatiion of full prolog*, 1995.

[CLLO95] Marshall P. CLINE and Greg A. LOMOW, *C++ Frequenctly Asked Questions*, Addison-Wesley Publiching Co., 1995.

[LCMR] LE CHARLIER B., CORSINI M., MUSUMBU K. and RAUZY A., *Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Symbolic Finite Domains.*

[LECH1L] LE CHARLIER B., lecture notes: *Théorie des langages: paradigmes de programmation*, 1° licence, 1993.

[LEC2L1] LE CHARLIER B., lecture notes: *Logic programming*, 2° licence, 1994.

[LEC2L2] LE CHARLIER B., lecture notes: *Computation logic*, 2° licence, 1994.

[LECH91] LE CHARLIER B., *L'analyse statique des programmes par interprétation abstraite*, Nouvelles de la Science et des Technologies vol. 9, number 4, 1991.

[LECH??] LE CHARLIER B., *Abstract Interpretation and Finite Domain Symbolic Constraints*,

[LVM91] LE CHARLIER B., MUSUMBU K. and VAN HENTENRYCK P., *A Generic Abstract Interpretation Algorithm and its Complexity Analysis*, 1991.

[LEVA94] LE CHARLIER B. and VAN HENTENRYCK P., *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, ACM Transactions on Programming Languages and Systems (TOPLAS), 1994.

[LVBM94] LE CHARLIER B., BRAEM C., MODART S. and VAN HENTENRYCK P., *Cardinality Analysis of Prolog*, 1994.

[LVC94] LE CHARLIER B., CORTESI A. and VAN HENTENRYCK P., *Evaluation of the domain Prop*, 1994.

[LVC294] LE CHARLIER B., CORTESI A. and VAN HENTENRYCK P., *Type Analysis of Prolog using Type Graphs*, 1994.

[PVH96] VAN HENTENRYCK P., personal communication.

[STSH86] L. STERLING and E. SHAPIRO, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, Ma, 1986.

[TEN] TENNENT R. D., Principles of Programming languages, Prentice Hall.

Annexes

Contents of the annexes:

A1. LISTINGS OF SOME BENCHMARK PROGRAMS

The programs we use are hopefully representative of « pure » logic programs. They are taken from a number of authors and used for various purposes as explained later.

A1.1. PEEPHOLE

Peephole is a program written by S. Debray to carry out the peephole optimization in the SB-Prolog compiler.

```
comp_peepopt(Pil,OptPil,Preds):-
     comp_popt1(Pil, Pil1),
     comp_popt4(Pil1,[],_,Preds,OptPil).

comp_popt1([], []).
comp_popt1([Inst|Rest], Pil1):- comp_popt11(Inst, Rest, Pil1).

comp_popt11_aux(T,R,Inst,PRest,OptInstList):-
      T = R, !,
        comp_popt11(Inst,PRest,OptInstList).
comp_popt11_aux(T,R,Inst,PRest,OptInstList):-
        popt_movreg(Inst,R,T,PRest,OptInstList).
comp_popt11_aux1(PRest,R,OptInstList,S,OptPRest):-
      peep_chk(PRest,R), !,
      OptInstList = [unitvar(S)|OptPRest].
comp_popt11_aux1(PRest,R,OptInstList,S,OptPRest):-
        OptInstList = [unitvar(R),movreg(R,S)|OptPRest].

comp_popt11(puttvar(T,R), [getstr(S,R)|PRest], [putstr(S,T)|OptPRest]):-
      !,
      comp_popt1a(PRest, OptPRest).
comp_popt11(puttvar(T,R), [getlist(R)|PRest], [putlist(T)|OptPRest]):-
      !,
      comp_popt1a(PRest, OptPRest).
comp_popt11(movreg(T,R),[Inst|PRest],OptInstList):-
      !,
        comp_popt11_aux(T,R,Inst,PRest,OptInstList).
comp_popt11(putpvar(V,R), [getpval(V,R)|PRest], [putpvar(V,R)|OptPRest]):-
      !,
      comp_popt1(PRest, OptPRest).
comp_popt11(putpvar(V,R), [getstr(Str,R)|PRest], [putstrv(Str,V)|OptPRest]):-
      !,
      comp_popt1a(PRest, OptPRest).
comp_popt11(putpval(V,R), [getstr(Str,R)|PRest], [getstrv(Str,V)|OptPRest]):-
      !,
      comp_popt1(PRest, OptPRest).
comp_popt11(getlist(R),
[unitvar(R1),unitvar(R2)|PRest],[getlist_tvar_tvar(R,R1,R2)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(getcomma(R),
[unitvar(R1),unitvar(R2)|PRest],[getcomma_tvar_tvar(R,R1,R2)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(getlist_k(R),
[unitvar(R1),unitvar(R2)|PRest],[getlist_k_tvar_tvar(R,R1,R2)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(gettval(R,R), PRest,OptPRest):-
      !,
      comp_popt1(PRest, OptPRest).
comp_popt11(unitvar(R), [movreg(R,S)|PRest], OptInstList):-
      !,
        comp_popt11_aux1(PRest,R,OptInstList,S,OptPRest),
      comp_popt1(PRest, OptPRest).
comp_popt11(jump(L), [label(L)|PRest],  [label(L)|OptPRest]):-
```

```
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(jump(Addr), [jump(_)|PRest],  [jump(Addr)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(jumpz(_,L), [label(L)|PRest],  [label(L)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(jumpnz(_,L), [label(L)|PRest],  [label(L)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(jumplt(_,L), [label(L)|PRest],  [label(L)|OptPRest]):-
      !,
      comp_popt1(PRest, OptPRest).
comp_popt11(jumple(_,L), [label(L)|PRest],  [label(L)|OptPRest]):-
      !,
      comp_popt1(PRest, OptPRest).
comp_popt11(jumpgt(_,L), [label(L)|PRest],  [label(L)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(jumpge(_,L), [label(L)|PRest],  [label(L)|OptPRest]):-
      !,
      comp_popt1(PRest,OptPRest).
comp_popt11(Inst, PRest, [Inst|OptPRest]):-
      comp_popt1(PRest, OptPRest).

comp_popt1a([], []).
comp_popt1a([Inst|PRest], OptPList):-
     popt_uni2bld(Inst,BldInst), ! ,
     OptPList = [BldInst|OptPRest],
     comp_popt11(Inst,PRest,OptPList).
comp_popt1a([Inst|PRest], OptPList):-
     comp_popt1a(PRest,OptPRest),
     comp_popt11(Inst,PRest,OptPList).

popt_uni2bld(unipvar(X), bldpvar(X)).
popt_uni2bld(unipval(X), bldpval(X)).
popt_uni2bld(unitvar(X), bldtvar(X)).
popt_uni2bld(unitval(X), bldtval(X)).
popt_uni2bld(unicon(X), bldcon(X)).
popt_uni2bld(uninil, bldnil).
popt_uni2bld(uninumcon(X), bldnumcon(X)).
popt_uni2bld(unifloatcon(X), bldfloatcon(X)).

comp_popt4_aux(El,OList,ORest,Inst):-
      El = 1, !,
      OList = ORest.
comp_popt4_aux(El,OList,ORest,Inst):-
      OList = [Inst|ORest].

comp_popt4([],_,_,Preds,[]).
comp_popt4([Inst|IRest],RCont,Seen,Preds,OList):-
      popt_builtin(Inst,Preds,OList,ORest), !,
      RCont1 = RCont,
      comp_popt4(IRest,RCont1,Seen,Preds,ORest).
comp_popt4([Inst|IRest],RCont,Seen,Preds,OList):-
      peep_redundant(Inst,IRest,RCont,RCont1,Seen,El),
        comp_popt4_aux(El,OList,ORest,Inst),
      comp_popt4(IRest,RCont1,Seen,Preds,ORest).

popt_builtin(call(P,N,_),Preds,[builtin(Bno)|IRest],IRest):-
       comp_builtin(P,N,Bno),
       not_member1(slash(P,N),Preds),
       !.
popt_builtin(calld(P,N,_),Preds,[builtin(Bno)|IRest],IRest):-
       comp_builtin(P,N,Bno),
       not_member1(slash(P,N),Preds),
       !.
popt_builtin(execute(comma(P,N)),Preds,[builtin(Bno),proceed|IRest],IRest):-
       comp_builtin(P,N,Bno),
       not_member1(slash(P,N),Preds).

popt_movreg(Inst,R,T,PRest,OptInstList):-
      popt_movreg0(Inst,R,T,OptInst),
        peep_chk(PRest,R), !,
      OptInstList = [OptInst|OptInstRest],
```

```
      comp_popt1(PRest, OptInstRest).
popt_movreg(Inst,R,T,PRest,OptInstList):-
      OptInstList = [movreg(T,R),Inst|OptInstRest],
      comp_popt1(PRest, OptInstRest).

popt_movreg0(getstr(S,R),R,T,getstr(S,T)).
popt_movreg0(puttbreg(R),R,T,puttbreg(T)).
popt_movreg0(addreg(R,S),R,T,addreg(T,S)).
popt_movreg0(subreg(R,S),R,T,subreg(T,S)).
popt_movreg0(mulreg(R,S),R,T,mulreg(T,S)).
popt_movreg0(divreg(R,S),R,T,divreg(T,S)).
popt_movreg0(idivreg(R,S),R,T,idivreg(T,S)).
popt_movreg0(get_tag(R,S),R,T,get_tag(T,S)).
popt_movreg0(arg(R,R2,R3),R,T,arg(T,R2,R3)).
popt_movreg0(arg(R1,R,R3),R,T,arg(R1,T,R3)).
popt_movreg0(arg(R1,R2,R),R,T,arg(R1,R2,T)).
popt_movreg0(arg0(R,R2,R3),R,T,arg0(T,R2,R3)).
popt_movreg0(arg0(R1,R,R3),R,T,arg0(R1,T,R3)).
popt_movreg0(arg0(R1,R2,R),R,T,arg0(R1,R2,T)).
popt_movreg0(test_unifiable(R,R2,R3),R,T,test_unifiable(T,R2,R3)).
popt_movreg0(test_unifiable(R1,R,R3),R,T,test_unifiable(R1,T,R3)).
popt_movreg0(test_unifiable(R1,R2,R),R,T,test_unifiable(R1,R2,T)).

popt_chkmember_aux(P,P1,Flag,L1):-
       P = P1, !,
       Flag = 0.
popt_chkmember_aux(P,P1,Flag,L1):-
       popt_chkmember(P,L1,Flag).

popt_chkmember(P,L,Flag):-
      var(L), !,
        L = [P|_],
      Flag = 1.
popt_chkmember(P,L,Flag):-
      nonvar(L),
        L = [P1|L1],
        popt_chkmember_aux(P,P1,Flag,L1).

peep_use(getcon(_,R),R).
peep_use(getnumcon(_,R),R).
peep_use(getfloatcon(_,R),R).
peep_use(getpval(_,R),R).
peep_use(gettval(_,R),R).
peep_use(gettval(R,_),R).
peep_use(gettbreg(R),R).
peep_use(getpbreg(R),R).
peep_use(getstr(_,R),R).
peep_use(getstrv(_,R),R).
peep_use(getlist(R),R).
peep_use(getlist_tvar_tvar(R,_,_),R).
peep_use(getcomma(R),R).
peep_use(getcomma_tvar_tvar(R,_,_),R).
peep_use(get_tag(R,_),R).
peep_use(unitval(R),R).
peep_use(unipval(R),R).
peep_use(bldtval(R),R).
peep_use(bldpval(R),R).
peep_use(arg(R,_,_),R).
peep_use(arg(_,R,_),R).
peep_use(arg(_,_,R),R).
peep_use(arg0(R,_,_),R).
peep_use(arg0(_,R,_),R).
peep_use(arg0(_,_,R),R).
peep_use(test_unifiable(R,_,_),R).
peep_use(test_unifiable(_,R,_),R).
peep_use(and(R,_),R).
peep_use(and(_,R),R).
peep_use(negate(R),R).
peep_use(or(R,_),R).
peep_use(or(_,R),R).
peep_use(lshiftl(R,_),R).
peep_use(lshiftl(_,R),R).
peep_use(lshiftr(R,_),R).
peep_use(lshiftr(_,R),R).
peep_use(addreg(R,_),R).
peep_use(addreg(_,R),R).
```

```
peep_use(subreg(R,_),R).
peep_use(subreg(_,R),R).
peep_use(mulreg(R,_),R).
peep_use(mulreg(_,R),R).
peep_use(divreg(R,_),R).
peep_use(divreg(_,R),R).
peep_use(idivreg(R,_),R).
peep_use(idivreg(_,R),R).
peep_use(movreg(R,_),R).
peep_use(switchonterm(R,_,_),R).
peep_use(switchonlist(R,_,_),R).
peep_use(switchonbound(R,_,_),R).
peep_use(jump(_),_).
peep_use(jumpeq(R,L),R):- L \== abs(-1).
peep_use(jumpne(R,L),R):- L \== abs(-1).
peep_use(jumplt(R,L),R):- L \== abs(-1).
peep_use(jumple(R,L),R):- L \== abs(-1).
peep_use(jumpgt(R,L),R):- L \== abs(-1).
peep_use(jumpge(R,L),R):- L \== abs(-1).

peep_chk([],_).
peep_chk([Inst|Rest],R):-
      peep_use(Inst,R), !, fail.
peep_chk([Inst|Rest],R):-
      peep_term(Inst,R),
        !.
peep_chk([Inst|Rest],R):-
        peep_chk(Rest,R).

peep_term(call(_,_,_),_).
peep_term(calld(_,_,_),_).
peep_term(execute(_),_).
peep_term('_execmarker',_).
peep_term(putcon(R),R).
peep_term(putnumcon(R),R).
peep_term(putfloatcon(R),R).
peep_term(puttvar(R,_),R).
peep_term(putpvar(_,R),R).
peep_term(putdval(_,R),R).
peep_term(putuval(_,R),R).
peep_term(puttbreg(R),R).
peep_term(putpval(_,R),R).
peep_term(putstr(_,R),R).
peep_term(putstrv(_,R),R).
peep_term(putlist(R),R).
peep_term(putnil(R),R).
peep_term(get_tag(_,R),R).
peep_term(movreg(_,R),R).
peep_term(bldtvar(R),R).
peep_term(test_unifiable(_,_,R),R).

peep_redundant('_execmarker',_,R,R,_,1).
peep_redundant(Inst,IRest,RCont,RCont1,Seen,El):-
      peep_elim(Inst,IRest,RCont,RCont1,Seen,El), !.
peep_redundant(Inst,IRest,RCont,RCont1,Seen,El):-
      RCont1 = RCont, El = 0.

peep_elim(jumpz(_,L),_,R0,R1,_,0):- L == abs(-1), !, R1 = R0.
peep_elim(jumpz(_,L),_,R0,R1,_,0):- R1 = [].
peep_elim(getpvar(V,R),_,RCont,[r(R,v(V))|RCont],_,0).
peep_elim(getpval(V,R),_,RCont,RCont1,Seen,El):-
      member1(r(R,v(V)),RCont), !,
      El = 1, RCont1 = Rcont.
peep_elim(getpval(V,R),_,RCont,RCont1,Seen,El):-
        El = 0, RCont1 = [r(R,v(V))|RCont].
peep_elim(getcon(C,R),_,RCont,RCont1,Seen,El):-
      member1(r(R,c(C)),RCont), !,
      El = 1, RCont1 = Rcont.
peep_elim(getcon(C,R),_,RCont,RCont1,Seen,El):-
      El = 0, RCont1 = [r(R,c(C))|RCont].
peep_elim(getnumcon(N,R),_,RCont,RCont1,Seen,El):-
      member1(r(R,n(N)),RCont), !,
      El = 1, RCont1 = Rcont.
peep_elim(getnumcon(N,R),_,RCont,RCont1,Seen,El):-
      El = 0, RCont1 = [r(R,n(N))|RCont].
peep_elim(getfloatcon(N,R),_,RCont,RCont1,Seen,El):-
```

```
      member1(r(R,nf(N)),RCont), !,
      El = 1, RCont1 = Rcont.
peep_elim(getfloatcon(N,R),_,RCont,RCont1,Seen,El):-
      El = 0, RCont1 = [r(R,nf(N))|RCont].
peep_elim(getnil(R),_,RCont,RCont1,Seen,El):-
      member1(r(R,c(nil)),RCont), !,
      El = 1, RCont1 = Rcont.
peep_elim(getnil(R),_,RCont,RCont1,Seen,El):-
      El = 0, RCont1 = [r(R,c(nil))|RCont].
peep_elim(putpvar(V,R),_,L0,L1,_,0):-
      peep_elim_upd(L0,R,v(V),L1).
peep_elim(putpval(V,R),_,RCont,RCont1,_,El):-
      member1(r(R,v(V)),RCont), !,
      El = 1, RCont1 = RCont.
peep_elim(putpval(V,R),_,RCont,RCont1,_,El):-
      El = 0, peep_elim_upd(RCont,R,v(V),RCont1).
peep_elim(puttvar(R,R1),_,L0,L1,_,0):-
      peep_del(L0,r(R,_),L2), peep_del(L2,r(R1,_),L1).
peep_elim(putcon(C,R),_,RCont,RCont1,_,El):-
      member1(r(R,c(C)),RCont), !,
      El = 1, RCont1 = RCont.
peep_elim(putcon(C,R),_,RCont,RCont1,_,E1):-
      E1 = 0, peep_elim_upd(RCont,R,c(C),RCont1).
peep_elim(putnumcon(N,R),_,RCont,RCont1,_,El):-
      member1(r(R,n(N)),RCont), !,
      El = 1, RCont1 = RCont.
peep_elim(putnumcon(N,R),_,RCont,RCont1,_,El):-
      El = 0, peep_elim_upd(RCont,R,n(N),RCont1).
peep_elim(putfloatcon(N,R),_,RCont,RCont1,_,El):-
      member1(r(R,nf(N)),RCont), !,
      El = 1, RCont1 = RCont.
peep_elim(putfloatcon(N,R),_,RCont,RCont1,_,El):-
      El = 0, peep_elim_upd(RCont,R,nf(N),RCont1).
peep_elim(putnil(R),_,RCont,RCont1,_,El):-
      member1(r(R,c(nil)),RCont), !,
      El = 1, RCont1 = RCont.
peep_elim(putni1(R),_,RCont,RCont1,_,El):-
      El = 0, peep_elim_upd(RCont,R,c(nil),RCont1).
peep_elim(putstr(F,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(putlist(R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(and(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(or(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(negate(R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(lshiftr(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(lshiftl(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(addreg(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(subreg(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(mulreg(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(divreg(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(idivreg(_,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(movreg(R,R1),_,L0,L1,_,0):- peep_elim_upd(L0,R1,r(R),L1).
peep_elim(gettbreg(R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(putdval(V,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(putuval(V,R),_,L0,L1,_,0):- peep_del(L0,r(R,_),L1).
peep_elim(label(comma(P,N,K)),_,_,[],Seen,0):-
      N >= 0, !, member1(comma(P,N),Seen).
peep_elim(label(comma(P,N,K)),_,_,[],Seen,0).
peep_elim(call(_,_,_),_,_,[],_,0).
peep_elim(proceed,_,_,[],_,0).
peep_elim(execute(comma(P,N)),IRest,_,[],Seen,El):-
      IRest = [label(comma(P,N,K))|_], N >= 0, !,
      popt_chkmember(comma(P,N),Seen,El).
peep_elim(execute(comma(P,N)),IRest,_,[],Seen,El):-
      El = 0.
peep_elim(calld(_,_,_),_,_,[],_,0).
peep_elim(builtin(_),_,_,[],_,0).
peep_elim(trymeelse(_,_),_,_,[],_,0).
peep_elim(retrymeelse(_,_),_,_,[],_,0).
peep_elim(trustmeelsefail(_),_,_,[],_,0).
peep_elim(try(_,_),_,_,[],_,0).
peep_elim(retry(_,_),_,_,[],_,0).
peep_elim(trust(_),_,_,[],_,0).
peep_elim(jump(_),_,_,[],_,0).
peep_elim(jumpnz(_,L),_,R0,R1,_,0):- L == abs(-1), !, R1 = R0.
peep_elim(jumpnz(_,L),_,R0,R1,_,0):- R1 = [].
peep_elim(jumplt(_,L),_,R0,R1,_,0):- L == abs(-1),!, R1 = R0.
```

```
peep_elim(jumplt(_,L),_,R0,R1,_,0):- R1 = [].
peep_elim(jumple(_,L),_,R0,R1,_,0):- L == abs(-1), !, R1 = R0.
peep_elim(jumple(_,L),_,R0,R1,_,0):- R1 = [].
peep_elim(jumpgt(_,L),_,R0,R1,_,0):- L == abs(-1), !, R1 = R0.
peep_elim(jumpgt(_,L),_,R0,R1,_,0):- R1 = [].
peep_elim(jumpge(_,L),_,R0,R1,_,0):- L == abs(-1), !, R1 = R0.
peep_elim(jumpge(_,L),_,R0,R1,_,0):- R1 = [].
peep_elim(switchonterm(_,_,_),_,_,[],_,0).
peep_elim(switchonlist(_,_,_),_,_,[],_,0).
peep_elim(switchonbound(_,_,_),_,_,[],_,0).

peep_del([],_,[]).
peep_del([X|L],Y,L1):-
      X == Y, !,
      L1 = L1Rest,
      peep_del(L,Y,L1Rest).
peep_del([X|L],Y,L1):-
      L1 = [X|L1Rest],
      peep_del(L,Y,L1Rest).

peep_elim_upd(L0,R,Cont,[r(R,Cont)|L1]):- peep_del(L0,r(R,_),L1).

comp_builtin(P,N,Bno):-
      ground(P),
      ground(N),
      ground(Bno).

not_member1(X,[]).
not_member1(X,[F|T]):-
   X \== F,
   not_member1(X,T).

member1(X,[X|Y]):- !.
member1(X,[F|T]):-
   member1(X,T).

member( X, [X|Xs]).
member( X, [_|Xs]):-
      member(X,Xs).

iota(N,List):-
      iota1(0,N,List).

iota1(K,K,[]):-!.
iota1(K,N,[K|List]):-
      K1:= K+1,
      iota1(K1,N,List).

dif( [],_,_,[],[]).
dif( [S|Ss],Val,Mod,[D|Ds],[D2|D2s]):-
     D:= Val - S,
     D2:= Mod - D,
     dif( Ss,Val,Mod,Ds,D2s).

rev( [],L,L).
rev( [X|Xs], Y,L):-rev( Xs,[X|Y],L).

mergedelete([],L,L).
mergedelete([D|Ds], [D|R], L2):-
        mergedelete( Ds, R, L2).
mergedelete( [D|Ds], [X|R], [X|L2]):-
        D > X,
        mergedelete( [D|Ds],R,L2).

check( [],_,L,L,_):-!.
check( S, Choice, Old, L3,Modulus):-
      dif( S, Choice,Modulus, Ds,Dds),
      mergedelete( Ds, Old,L2),
      rev( Dds,[],Rds),
      mergedelete(Rds, L2,L3),
      !.

pds1( [],_,[],_):-!.
pds1( Unused, List, [Choice|Rest],Mod):-
     member(Choice,Unused),
     check( List, Choice, Unused,U3,Mod),
```

```
     pds1( U3, [Choice| List], Rest, Mod).

pds( Order, [0|Ans]):-
    N:= Order * (Order + 1) + 1,
    iota( N, [0|List]),
    pds1( List, [0], Ans, N).

pdsbm(N):- pds(N, [0,1|X]).
```

A1.2. READ

The program *read* is the tokeniser and reader written by R. O'keefe and D.H.D. Warren for Prolog.

```
read(Answer, Variables):-
      repeat,
      read_tokens(Tokens, Variables),
      read_aux(Tokens),!,
      Answer = Term.

read_aux(Tokens):-
      read(Tokens, 1200, Term, LeftOver), all_read(LeftOver), !.
read_aux(Tokens):-
      syntax_error(Tokens,X).

repeat.
repeat:-
      repeat.

all_read([]):- !.
all_read(S):-
      syntax_error([operator,expected,after,expression], S).

expect(Token, [Token|Rest], Rest):- !.
expect(Token, S0, _):-
      syntax_error([Token,or,operator,expected], S0).

prefixop(Op, Prec, Prec):-
      current_op(Prec, fy, Op), !.
prefixop(Op, Prec, Less):-
      current_op(Prec, fx, Op), !,
      Less:= Prec-1.

postfixop(Op, Prec, Prec):-
      current_op(Prec, yf, Op), !.
postfixop(Op, Less, Prec):-
      current_op(Prec, xf, Op), !, Less:= Prec-1.

infixop(Op, Less, Prec, Less):-
      current_op(Prec, xfx, Op), !, Less:= Prec-1.
infixop(Op, Less, Prec, Prec):-
      current_op(Prec, xfy, Op), !, Less:= Prec-1.
infixop(Op, Prec, Prec, Less):-
      current_op(Prec, yfx, Op), !, Less:= Prec-1.

ambigop(F, L1, O1, R1, L2, O2):-
      postfixop(F, L2, O2),
      infixop(F, L1, O1, R1), !.

read([Token|RestTokens], Precedence, Term, LeftOver):-
      read(Token, RestTokens, Precedence, Term, LeftOver).
read([], _, _, _):-
      syntax_error([expression,expected], []).
read(var(Variable,_), ['('|S1], Precedence, Answer, S):- !,
      read(S1, 999, Arg1, S2),
      read_args(S2, RestArgs, S3), !,
      exprtl0(S3, apply(Variable,[Arg1|RestArgs]), Precedence, Answer, S).
read(var(Variable,_), S0, Precedence, Answer, S):- !,
      exprtl0(S0, Variable, Precedence, Answer, S).
read(atom('-'), [integer(Integer)|S1], Precedence, Answer, S):-
      Negative:= -Integer, !,
```

```
      exprtl0(S1, Negative, Precedence, Answer, S).
read(atom(Functor), ['('|S1], Precedence, Answer, S):- !,
      read(S1, 999, Arg1, S2),
      read_args(S2, RestArgs, S3),
      univ(Term,[Functor,Arg1|RestArgs]), !,
      exprtl0(S3, Term, Precedence, Answer, S).
read(atom(Functor), S0, Precedence, Answer, S):-
      prefixop(Functor, Prec, Right), !,
      after_prefix_op(Functor, Prec, Right, S0, Precedence, Answer, S).
read(atom(Atom), S0, Precedence, Answer, S):- !,
      exprtl0(S0, Atom, Precedence, Answer, S).
read(integer(Integer), S0, Precedence, Answer, S):- !,
      exprtl0(S0, Integer, Precedence, Answer, S).
read('[', [']'|S1], Precedence, Answer, S):- !,
      exprtl0(S1, [], Precedence, Answer, S).
read('[', S1, Precedence, Answer, S):- !,
      read(S1, 999, Arg1, S2),
      read_list(S2, RestArgs, S3), !,
      exprtl0(S3, [Arg1|RestArgs], Precedence, Answer, S).
read('(', S1, Precedence, Answer, S):- !,
      read(S1, 1200, Term, S2),
      expect(')', S2, S3), !,
      exprtl0(S3, Term, Precedence, Answer, S).
read(' (', S1, Precedence, Answer, S):- !,
      read(S1, 1200, Term, S2),
      expect(')', S2, S3), !,
      exprtl0(S3, Term, Precedence, Answer, S).
read('{', ['}'|S1], Precedence, Answer, S):- !,
      exprtl0(S1, '{}', Precedence, Answer, S).
read('{', S1, Precedence, Answer, S):- !,
      read(S1, 1200, Term, S2),
      expect('}', S2, S3), !,
      exprtl0(S3, '{}', Precedence, Answer, S).
read(string(List), S0, Precedence, Answer, S):- !,
      exprtl0(S0, List, Precedence, Answer, S).
read(Token, S0, _, _, _):-
      syntax_error([Token,cannot,start,an,expression], S0).

read_args([','|S1], [Term|Rest], S):- !,
      read(S1, 999, Term, S2), !,
      read_args(S2, Rest, S).
read_args([')'|S], [], S):- !.
read_args(S, _, _):-
      syntax_error([', or )',expected,in,arguments], S).

read_list([','|S1], [Term|Rest], S):- !,
      read(S1, 999, Term, S2), !,
      read_list(S2, Rest, S).
read_list(['|'|S1], Rest, S):- !,
      read(S1, 999, Rest, S2), !,
      expect(']', S2, S).
read_list([']'|S], [], S):- !.
read_list(S, _, _):-
      syntax_error([', | or ]',expected,in,list], S).

after_prefix_op(Op, Oprec, Aprec, S0, Precedence, _, _):-
      Precedence < Oprec, !,
      syntax_error([prefix,operator,Op,in,context,
            with,precedence,Precedence], S0).
after_prefix_op(Op, Oprec, Aprec, S0, Precedence, Answer, S):-
      peepop(S0, S1),
      prefix_is_atom(S1, Oprec),
      exprtl(S1, Oprec, Op, Precedence, Answer, S).
after_prefix_op(Op, Oprec, Aprec, S1, Precedence, Answer, S):-
      read(S1, Aprec, Arg, S2),
      univ(Term,[Op,Arg]), !,
      exprtl(S2, Oprec, Term, Precedence, Answer, S).

peepop([atom(F),'('|S1], [atom(F),'('|S1]):- !.
peepop([atom(F)|S1], [infixop(F,L,P,R)|S1]):- infixop(F, L, P, R).
peepop([atom(F)|S1], [postfixop(F,L,P)|S1]):- postfixop(F, L, P).
peepop(S0, S0).

prefix_is_atom([Token|_], Precedence):-
      prefix_is_atom(Token, Precedence).
prefix_is_atom(infixop(_,L,_,_), P):- L >= P.
```

```
prefix_is_atom(postfixop(_,L,_), P):- L >= P.
prefix_is_atom(')', _).
prefix_is_atom(']', _).
prefix_is_atom('}', _).
prefix_is_atom('|', P):- 1100 >= P.
prefix_is_atom(',', P):- 1000 >= P.
prefix_is_atom([],  _).

exprtl0([atom(F)|S1], Term, Precedence, Answer, S):-
      ambigop(F, L1, O1, R1, L2, O2), !,
      exprtl([infixop(F,L1,O1,R1)|S1], 0, Term, Precedence, Answer, S).
exprtl0([atom(F)|S1], Term, Precedence, Answer, S):-
      ambigop(F, L1, O1, R1, L2, O2), !,
      exprtl([postfixop(F,L2,O2) |S1], 0, Term, Precedence, Answer, S).
exprtl0([atom(F)|S1], Term, Precedence, Answer, S):-
      infixop(F, L1, O1, R1), !,
      exprtl([infixop(F,L1,O1,R1)|S1], 0, Term, Precedence, Answer, S).
exprtl0([atom(F)|S1], Term, Precedence, Answer, S):-
      postfixop(F, L2, O2), !,
      exprtl([postfixop(F,L2,O2) |S1], 0, Term, Precedence, Answer, S).
exprtl0([','|S1], Term, Precedence, Answer, S):-
      Precedence >= 1000, !,
      read(S1, 1000, Next, S2), !,
      exprtl(S2, 1000, comma(Term,Next), Precedence, Answer, S).
exprtl0(['|'|S1], Term, Precedence, Answer, S):-
      Precedence >= 1100, !,
      read(S1, 1100, Next, S2), !,
      exprtl(S2, 1100, comma(Term,Next), Precedence, Answer, S).
exprtl0([Thing|S1], _, _, _, _):-
      cant_follow_expr(Thing, Culprit), !,
      syntax_error([Culprit,follows,expression], [Thing|S1]).

exprtl0(S, Term, _, Term, S).

cant_follow_expr(atom(_),    atom).
cant_follow_expr(var(_,_),   variable).
cant_follow_expr(integer(_),integer).
cant_follow_expr(string(_), string).
cant_follow_expr(' (',              bracket).
cant_follow_expr('(',               bracket).
cant_follow_expr('[',               bracket).
cant_follow_expr('{',               bracket).

exprtl([infixop(F,L,O,R)|S1], C, Term, Precedence, Answer, S):-
      Precedence >= O, C <= L, !,
      read(S1, R, Other, S2),
      univ(Expr,[F,Term,Other]),
      exprtl(S2, O, Expr, Precedence, Answer, S).
exprtl([postfixop(F,L,O)|S1], C, Term, Precedence, Answer, S):-
      Precedence >= O, C <= L, !,
      univ(Expr,[F,Term]),
      peepop(S1, S2),
      exprtl(S2, O, Expr, Precedence, Answer, S).
exprtl([','|S1], C, Term, Precedence, Answer, S):-
      Precedence >= 1000, C < 1000, !,
      read(S1, 1000, Next, S2),
      exprtl(S2, 1000, comma(Term,Next), Precedence, Answer, S).
exprtl(['|'|S1], C, Term, Precedence, Answer, S):-
      Precedence >= 1100, C < 1100, !,
      read(S1, 1100, Next, S2),
      exprtl(S2, 1100, comma(Term,Next), Precedence, Answer, S).
exprtl(S, _, Term, _, Term, S).

syntax_error(Message, List):-
      fail.

univ(X,Y):-
      compound(X), !,
      functor(X,F,A),
      Y = [F|Args],
      collect_args(X,1,A,Args).
univ(X,L):-
      nonvar(L),
      !,
      L = [F|Args],
      nonvar(Args),
```

```
      lengthnovar(Args,A),
      functor(X,F,A),
      collect_args(X,1,A,Args).

collect_args(X,A1,A,[]):-
      A1 > A, !.
collect_args(X,A1,A,[T|Ts]):-
      arg(A1,X,T),
      A2:= A1 + 1,
      collect_args(X,A2,A,Ts).

lengthnovar([],0).
lengthnovar([F|T],N):-
      nonvar(T),
      lengthnovar(T,N1),
      N:= N1 + 1.

read_tokens(TokenList, Dictionary):-
      read_tokens(32, Dict, ListOfTokens),
      append(Dict, [], Dict), !,
      Dictionary = Dict,
      TokenList = ListOfTokens.
read_tokens([atom(end_of_file)], []).

p(26).
p(31).

read_tokens(-1, _, _):- !,
      fail.
read_tokens(Ch, Dict, Tokens):-
      Ch <= 32,
      !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(37, Dict, Tokens):- !,
      repeat,
      get0(Ch),
      p(Ch),
      !,
      Ch <> 2,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(47, Dict, Tokens):- !,
      get0(NextCh),
      read_solidus(NextCh, Dict, Tokens).
read_tokens(33, Dict, [atom(!)|Tokens]):- !,
      get0(NextCh),
      read_after_atom(NextCh, Dict, Tokens).
read_tokens(40, Dict, [' ('|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(41, Dict, [')'|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(44, Dict, [','|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(59, Dict, [atom(';')|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(91, Dict, ['['|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(93, Dict, [']'|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(123, Dict, ['{'|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(124, Dict, ['|'|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(125, Dict, ['}'|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(46, Dict, Tokens):- !,
```

```
      get0(NextCh),
      read_fullstop(NextCh, Dict, Tokens).
read_tokens(34, Dict, [string(S)|Tokens]):- !,
      read_string(S, 34, NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(39, Dict, [atom(A)|Tokens]):- !,
      read_string(S, 39, NextCh),
      ground(A),
      read_after_atom(NextCh, Dict, Tokens).
read_tokens(Ch, Dict, [var(Var,Name)|Tokens]):-
      Ch >= 65, Ch <= 90,
      !,
      read_name(Ch, S, NextCh),
      ground(Name),
        read_lookup(Dict, Name=Var),
      !,
      read_tokens(NextCh, Dict, Tokens).
read_tokens(Ch, Dict, [integer(I)|Tokens]):-
      Ch >= 48, Ch <= 57,
      !,
      read_integer(Ch, I, NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_tokens(Ch, Dict, [atom(A)|Tokens]):-
      Ch >= 97, Ch <= 122,
      !,
      read_name(Ch, S, NextCh),
      ground(A),
      read_after_atom(NextCh, Dict, Tokens).
read_tokens(Ch, Dict, [atom(A)|Tokens]):-
      get0(AnotherCh),
      read_symbol(AnotherCh, Chars, NextCh),
      ground(A),
      read_after_atom(NextCh, Dict, Tokens).

read_after_atom(40, Dict, ['('|Tokens]):- !,
      get0(NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_after_atom(Ch, Dict, Tokens):-
      read_tokens(Ch, Dict, Tokens).

read_string(Chars, Quote, NextCh):-
      get0(Ch),
      read_string(Ch, Chars, Quote, NextCh).

read_string(26, _, Quote, 26):-
      display('! end of file in '), ttyput(Quote),
      display(token), ttyput(Quote), ttynl,
      !, fail.
read_string(Quote, Chars, Quote, NextCh):- !,
      get0(Ch),
      more_string(Ch, Quote, Chars, NextCh).
read_string(Char, [Char|Chars], Quote, NextCh):-
      read_string(Chars, Quote, NextCh).

more_string(Quote, Quote, [Quote|Chars], NextCh):- !,
      read_string(Chars, Quote, NextCh).
more_string(NextCh, _, [], NextCh).

read_solidus(42, Dict, Tokens):- !,
      get0(Ch),
      read_solidus(Ch, NextCh),
      read_tokens(NextCh, Dict, Tokens).
read_solidus(Ch, Dict, [atom(A)|Tokens]):-
      read_symbol(Ch, Chars, NextCh),
      ground(A),
      read_tokens(NextCh, Dict, Tokens).

read_solidus(26, 26):- !,
      display('! end of file in comment'), ttynl.
read_solidus(42, LastCh):-
      get0(NextCh),
      NextCh <> 47, !,
      read_solidus(NextCh, LastCh).
read_solidus(42, 32):- !.
read_solidus(_, LastCh):-
      get0(NextCh),
```

```
      read_solidus(NextCh, LastCh).

read_name(Char, [Char|Chars], LastCh):-
      Char >= 97, Char <= 122,
      !,
      get0(NextCh),
      read_name(NextCh, Chars, LastCh).
read_name(LastCh, [], LastCh).

current_op(X,Y,Z):- ground(X), ground(Y), ground(Z).

display(X).
ttyput(X).
ttynl.
get0(X):- ground(X).

read_symbol(Char, [Char|Chars], LastCh):-
      check_special(Char),
      get0(NextCh),
      read_symbol(NextCh, Chars, LastCh).
read_symbol(LastCh, [], LastCh).

check_special('#').
check_special('$').
check_special('&').
check_special('*').
check_special('+').
check_special('-').
check_special('.').
check_special('/').
check_special(':').
check_special('<').
check_special('=').
check_special('>').
check_special('?').
check_special('@').
check_special('\').
check_special('^').
check_special('`').
check_special('~').

read_fullstop(26, _, _):- !,
      display('! end of file just after full stop'), ttynl,
      fail.
read_fullstop(Ch, _, []):-
      Ch <= 32, !.
read_fullstop(Ch, Dict, [atom(A)|Tokens]):-
      read_symbol(Ch, S, NextCh),
      ground(A),
      read_tokens(NextCh, Dict, Tokens).

read_integer(BaseChar, IntVal, NextCh):-
      Base:= BaseChar - 48,
      get0(Ch),
      Ch <> 26,
      Ch <> 39, read_digits(Ch, Base, 10, IntVal, NextCh), !.
read_integer(BaseChar, IntVal, NextCh):-
      Base:= BaseChar - 48,
      get0(Ch),
      Ch <> 26,
      Base >= 1, read_digits(0, Base, IntVal, NextCh),
      !.
read_integer(BaseChar, IntVal, NextCh):-
      Base:= BaseChar - 48,
      get0(Ch),
      Ch <> 26,
      get0(IntVal), IntVal <> 26, get0(NextCh),
      !.

read_digits(SoFar, Base, Value, NextCh):-
      get0(Ch),
      Ch <> 26,
      read_digits(Ch, SoFar, Base, Value, NextCh).

read_digits(Digit, SoFar, Base, Value, NextCh):-
      Digit >= 48, Digit <= 57,
```

```
      !,
      Next:= SoFar*Base-48+Digit,
      read_digits(Next, Base, Value, NextCh).
read_digits(LastCh, Value, _, Value, LastCh).

read_lookup([X|_], X):- !.
read_lookup([_|T], X):-
      read_lookup(T, X).

append([],L,L).
append([H|L1],L2,[H|L3]):- append(L1,L2,L3).
```

A1.3. KALAH_R

The program *kalah* is a program which plays the game of kalah. This program is taken from [STSH86].

```
play(Game,Result):-
   initialize(Game,Position,Player),
   displaygame(Position,Player),
   play(Position,Player,Result).

play(Position,Player,Result):-
  gameover(Position,Player,Result),!,
  announce(Result).
play(Position,Player,Result):-
  choosemove(Position,Player,Move),
  move(Move,Position,Position1),
  displaygame(Position1,Player),
  nextplayer(Player,Player1), !,
  play(Position1,Player1,Result).

choosemove(Position,computer,Move):-
   lookahead(Depth),
   alphabeta(Depth,Position,40,40,Move,Value).
choosemove(Position,opponent,Move):-
   ground(Move),
   genlegal(Move).

alphabeta(0,Position,Alpha,Beta,Move,Value):-
   value(Position,Value).
alphabeta(D,Position,Alpha,Beta,Move,Value):-
  D > 0,
  allmoves(Position,Moves),
  Alpha1:= 0 - Beta,
  Beta1:= 0 - Alpha,
  D1:= D - 1,
  evaluateandchoose(Moves,Position,D1,Alpha1,Beta1,nil,p(Move,Value)).

allmoves(P,Res):-
  allmoves(P,[],Res).

allmoves(P,Acc,Res):-
  move(P,X),
  notmember(X,Acc), !,
  allmoves(P,[X|Acc],Res).
allmoves(P,Res,Res).

notmember(X,[]).
notmember(X,[F|T]):-
   X \== F,
   notmember(X,T).

evaluateandchoose([Move|Moves],Position,D,Alpha,Beta,Record,BestMove):-
  move(Move,Position,Position1),
  alphabeta(D,Position1,Alpha,Beta,MoveX,Value),
  Value1:= 0 - Value,
  cutoff(Move,Value1,D,Alpha,Beta,Moves,Position,Record,BestMove), !.
evaluateandchoose([],Position,D,Alpha,Beta,Move,p(Move,Alpha)).

cutoff(Move,Value,D,Alpha,Beta,Moves,Position,Move1,p(Move,Value)):-
```

```
  Value >= Beta, !.
cutoff(Move,Value,D,Alpha,Beta,Moves,Position,Move1,BestMove):-
  Alpha < Value,
  Value < Beta, !,
  evaluateandchoose(Moves,Position,D,Value,Beta,Move,BestMove).
cutoff(Move,Value,D,Alpha,Beta,Moves,Position,Move1,BestMove):-
  Value <= Alpha, !,
  evaluateandchoose(Moves,Position,D,Alpha,Beta,Move1,BestMove).

move(Board,[M|Ms]):-
   member(M,[1,2,3,4,5,6]),
   stonesinhole(M,Board,N),
   extendmove(N,M,Board,Ms).
move(board([0,0,0,0,0,0],K,Ys,L),[]).

member(X,[X|Y]).
member(X,[F|T]):-
   member(X,T).

stonesinhole(M,board(Hs,K,Ys,L),Stones):-
  nthmember(M,Hs,Stones),
  Stones > 0.

extendmove(Stones,M,Board,[]):-
  Stones <> 7 - M, !.
extendmove(Stones,M,Board,Ms):-
  Stones =:= 7 - M, !,
  distributestones(Stones,M,Board,Board1),
  move(Board1,Ms).

move([N|Ns],Board,FinalBoard):-
  stonesinhole(N,Board,Stones),
  distributestones(Stones,N,Board,Board1),
  move(Ns,Board1,FinalBoard).
move([],Board1,Board2):-
  swap(Board1,Board2).

distributestones(Stones,Hole,Board,FinalBoard):-
  distributemyholes(Stones,Hole,Board,Board1,Stones1),
  distributeyourholes(Stones1,Board1,FinalBoard).

distributemyholes(Stones,N,board(Hs,K,Ys,L),board(Hs1,K1,Ys,L),Stones1):-
  Stones > 7 - N, !,
  pickupanddistribute(N,Stones,Hs,Hs1),
  K1:= K + 1,
  Stones1:= Stones + N - 7.
distributemyholes(Stones,N,board(Hs,K,Ys,L),Board,0):-
  pickupanddistribute(N,Stones,Hs,Hs1),
  checkcapture(N,Stones,Hs1,Hs2,Ys,Ys1,Pieces),
  updatekalah(Pieces,N,Stones,K,K1),
  checkiffinished(board(Hs2,K1,Ys1,L),Board).

checkcapture(N,Stones,Hs,Hs1,Ys,Ys1,Pieces):-
  FinishingHole:= N + Stones,
  OppositeHole:= 7 - FinishingHole,
  nthmember(OppositeHole,Ys,Y),
  Y > 0, !,
  nsubstitute(OppositeHole,Hs,0,Hs1),
  nsubstitute(FinishingHole,Ys,0,Ys1),
  Pieces:= Y + 1.
checkcapture(N,Stones,Hs,Hs,Ys,Ys,0):- !.

checkiffinished(board(Hs,K,Ys,L),board(Hs,K,Hs,L1))  :-
 zero(Hs), !,
 sumlist(Ys,YsSum),
 L1:= L + YsSum.
checkiffinished(board(Hs,K,Ys,L),board(Ys,K1,Ys,L))  :-
 zero(Ys), !,
 sumlist(Hs,HsSum),
 K1:= K + HsSum.
checkiffinished(Board,Board):- !.

updatekalah(0,Stones,N,K,K):-
  Stones < 7 - N, !.
updatekalah(0,Stones,N,K,K1):-
  Stones =:= 7 - N, !,
```

```
  K1:= K + 1.
updatekalah(Pieces,Stones,N,K,K1):-
  Pieces > 0, !,
  K1:= K + Pieces.

distributeyourholes(0,Board,Board):- !.
distributeyourholes(Stones,board(Hs,K,Ys,L),board(Hs,K,Ys1,L)):-
   1 <= Stones,
   Stones <= 6,
   nonzero(Hs),!,
   distribute(Stones,Ys,Ys1).
distributeyourholes(Stones,board(Hs,K,Ys,L),board(Hs,K,Ys1,L)):-
   Stones > 6, !,
   distribute(6,Ys,Ys1),
   Stones1:= Stones - 6,
   distributestones(Stones1,1,board(Hs,K,Ys1,L),Board).
distributeyourholes(Stones,board(Hs,K,Ys,L),board(Hs,K,Hs,L1)):-
   zero(Hs), !,
   sumlist(Ys,YsSum),
   L1:= Stones + YsSum + L.

pickupanddistribute(1,N,[H|Hs],[0|Hs1]):-
   !, distribute(N,Hs,Hs1).
pickupanddistribute(K,N,[H|Hs],[0|Hs1]):-
   K > 1, !,
   K1:= K - 1,
   pickupanddistribute(K1,N,Hs,Hs1).

distribute(0,Hs,Hs):- !.
distribute(N,[H|Hs],[H1|Hs1]):-
   N > 0, !,
   N1:= N - 1,
   H1:= H + 1,
   distribute(N1,Hs,Hs1).
distribute(N,[],[]):- !.

value(board(H,K,Y,L),Value):-
  Value:= K - L.

gameover(board(N,0,N,0),Player,draw):-
  pieces(K),
  N =:= 6 * K, !.
gameover(board(H,K,Y,L),Player,Player):-
  pieces(N),
  K > 6 * N, !.
gameover(board(H,K,Y,L),Player,Opponent):-
  pieces(N),
  L > 6 * N,
  nextplayer(Player,Opponent).

announce(opponent).
announce(computer).
announce(draw).

nthmember(N,[H|Hs],K):-
  N > 1, !,
  N1:= N - 1,
  nthmember(N1,Hs,K).
nthmember(1,[H|Hs],H).

nsubstitute(1,[X|Xs],Y,[Y|Xs]):- !.
nsubstitute(N,[X|Xs],Y,[X|Xs1]):-
   N > 1,!,
   N1:= N -1,
   nsubstitute(N1,Xs,Y,Xs1).

nextplayer(computer,opponent).
nextplayer(opponent,computer).

legal([N|Ns]):-
  0 < N,
  N < 7,
  legal(Ns).
legal([]).

genlegal([N|Ns]):-
```

```
  member(N,[1,2,3,4,5,6]),
  genlegal(Ns).
genlegal([]).

swap(board(Hs,K,Ys,L),board(Ys,L,Hs,K)).

displaygame(Position,computer):-
  show(Position).
displaygame(Position,opponent):-
  swap(Position,Position1),
  show(Position1).

show(board(H,K,Y,L)):-
   reverse(H,Hr),
   writestones(Hr),
   writekalahs(K,L),
   writestones(Y).

writestones(H):-
  displayholes(H).

displayholes([H|Hs]):-
   writepile(H),
   displayholes(Hs).
displayholes ([]).

writepile(N):-
  N < 10,
  write(N).
writepile(N):-
  N > 10,
  write(N).

write(X).

writekalahs(K,L):-
  write(K),
  write(L).

zero([0,0,0,0,0,0]).
nonzero(Hs):-
  Hs \== [0,0,0,0,0,0].

reverse(L,K):-
   rev(L,[],K).

rev([],L,L).
rev([H|T],L,K):-
   rev(T,[H|L],K).

sumlist(Is,Sum):-
   sumlist(Is,0,Sum).
sumlist([],Sum,Sum).
sumlist([I|Is],Temp,Sum):-
  Temp1:= Temp + 1,
  sumlist(Is,Temp1,Sum).

lookahead(X):- ground(X).

initialize(X,Y,Z):- ground(X), ground(Y), ground(Z).

pieces(X):- ground(X).
```

A1.4. PRESS

The program *press* is an equation-solver program. This program is taken from [STSH86].

```
p(Z,Y,X):- call(test_press(X,Y)).

solve_equation(A*B=0,X,Solution):-
```

```
      !,
        factorize(A*B,X,Factors - []),
        remove_duplicates(Factors,Factors1),
        solve_factors(Factors1,X,Solution).

solve_equation(Equation,X,Solution):-
        single_occurrence(X,Equation),
      !,
        position(X,Equation,[Side|Position]),
        maneuver_sides(Side,Equation,Equation1),
        isolate(Position,Equation1,Solution).
solve_equation(Lhs=Rhs,X,Solution):-
        is_polynomial(Lhs,X),
        is_polynomial(Rhs,X),
      !,
        polynomial_normal_form(Lhs-Rhs,X,PolyForm),
        solve_polynomial_equation(PolyForm,X,Solution).
solve_equation(Equation,X,Solution):-
        offenders(Equation,X,Offenders),
        multiple(Offenders),
        homogenize(Equation,X,Offenders,Equation1,X1),
      !,
        solve_equation(Equation1,X1,Solution1),
        solve_equation(Solution1,X,Solution).

factorize(A*B,X,Factors - Rest):-
      !,
        factorize(A,X,Factors - Factors1),
        factorize(B,X,Factors1 - Rest).
factorize(C,X,[C|Factors] - Factors):-
        subterm(X,C), !.
factorize(C,X,Factors - Factors).

solve_factors([Factor|Factors],X,Solution):-
        solve_equation(Factor=0,X,Solution).
solve_factors([Factor|Factors],X,Solution):-
        solve_factors(Factors,X,Solution).

single_occurrence(Subterm,Term):-
        occurrence(Subterm,Term,1).

maneuver_sides(1,Lhs = Rhs,Lhs = Rhs):- !.
maneuver_sides(2,Lhs = Rhs,Rhs = Lhs):- !.

isolate([N|Position],Equation,IsolatedEquation):-
        isolax(N,Equation,Equation1),
        isolate(Position,Equation1,IsolatedEquation).
isolate([],Equation,Equation).

isolax(1,Term1+Term2 = Rhs,Term1 = Rhs-Term2).
isolax(2,Term1+Term2 = Rhs,Term2 = Rhs-Term1).

isolax(1,Term1-Term2 = Rhs,Term1 = Rhs+Term2).
isolax(2,Term1-Term2 = Rhs,Term2 = Term1-Rhs).

isolax(1,Term1*Term2 = Rhs,Term1 = Rhs/Term2):-
      Term2 <> 0.
isolax(2,Term1*Term2 = Rhs,Term2 = Rhs/Term1):-
      Term1 <> 0.

isolax(1,Term1/Term2 = Rhs,Term1 = Rhs*Term2):-
      Term2 <> 0.
isolax(2,Term1/Term2 = Rhs,Term2 = Term1/Rhs):-
      Rhs <> 0.

isolax(1,Term1^Term2 = Rhs,Term1 = Rhs^(Term2)).
isolax(2,Term1^Term2 = Rhs,Term2 = log(base(Term1),Rhs)).
isolax(1,sin(U) = V,U = arcsin(V)).
isolax(1,sin(U) = V,U = 180 - arcsin(V)).
isolax(1,cos(U) = V,U = arccos(V)).
isolax(1,cos(U) = V,U = arccos(V)).

natural_number(N):- N > 0.

is_polynomial(X,X):- !.
is_polynomial(Term,X):-
```

```
      constant(Term), !.
is_polynomial(Term1+Term2,X):- !,
        is_polynomial(Term1,X),
        is_polynomial(Term2,X).
is_polynomial(Term1-Term2,X):- !,
        is_polynomial(Term1,X),
        is_polynomial(Term2,X).
is_polynomial(Term1*Term2,X):- !,
        is_polynomial(Term1,X),
        is_polynomial(Term2,X).
is_polynomial(Term1/Term2,X):- !,
      is_polynomial(Term1,X),
        constant(Term2).
is_polynomial(Term^N,X):- !,
        natural_number(N),
      is_polynomial(Term,X).

polynomial_normal_form(Polynomial,X,NormalForm):-
        polynomial_form(Polynomial,X,PolyForm),
        remove_zero_terms(PolyForm,NormalForm).

polynomial_form(X,X,[p(1,1)]).
      polynomial_form(X^N,X,[p(1,N)]).
polynomial_form(Term1+Term2,X,PolyForm):-
        polynomial_form(Term1,X,PolyForm1),
        polynomial_form(Term2,X,PolyForm2),
        add_polynomials(PolyForm1,PolyForm2,PolyForm).
polynomial_form(Term1-Term2,X,PolyForm):-
        polynomial_form(Term1,X,PolyForm1),
        polynomial_form(Term2,X,PolyForm2),
        subtract_polynomials(PolyForm1,PolyForm2,PolyForm).
polynomial_form(Term1*Term2,X,PolyForm):-
        polynomial_form(Term1,X,PolyForm1),
        polynomial_form(Term2,X,PolyForm2),
        multiply_polynomials(PolyForm1,PolyForm2,PolyForm).
polynomial_form(Term^N,X,PolyForm):- !,
        polynomial_form(Term,X,PolyForm1),
        binomial(PolyForm1,N,PolyForm).
polynomial_form(Term,X,[p(Term,0)]):-
        free_of(X,Term), !.

remove_zero_terms([p(0,N)|Poly],Poly1):- !,
         remove_zero_terms(Poly,Poly1).
remove_zero_terms([p(C,N)|Poly],[p(C,N)|Poly1]):-
        C <> 0, !, remove_zero_terms(Poly,Poly1).
remove_zero_terms([],[]).

add_polynomials([],Poly,Poly):- !.
add_polynomials(Poly,[],Poly):- !.
add_polynomials([p(Ai,Ni)|Poly1],[p(Aj,Nj)|Poly2],[p(Ai,Ni)|Poly]):-
        Ni > Nj, !, add_polynomials(Poly1,[p(Aj,Nj)|Poly2],Poly).
add_polynomials([p(Ai,Ni)|Poly1],[p(Aj,Nj)|Poly2],[p(A,Ni)|Poly]):-
        Ni =:= Nj, !, A:= Ai+Aj, add_polynomials(Poly1,Poly2,Poly).
add_polynomials([p(Ai,Ni)|Poly1],[p(Aj,Nj)|Poly2],[p(Aj,Nj)|Poly]):-
        Ni < Nj, add_polynomials([p(Ai,Ni)|Poly1],Poly2,Poly).

subtract_polynomials(Poly1,Poly2,Poly):-
        multiply_single(Poly2,p(1,0),Poly3),
        add_polynomials(Poly1,Poly3,Poly), !.

multiply_single([p(C1,N1)|Poly1],p(C,N),[p(C2,N2)|Poly]):-
        C2:= C1*C, N2:= N1+N, multiply_single(Poly1,p(C,N),Poly).
multiply_single([],Factor,[]).

multiply_polynomials([p(C,N)|Poly1],Poly2,Poly):-
        multiply_single(Poly2,p(C,N),Poly3),
        multiply_polynomials(Poly1,Poly2,Poly4),
           add_polynomials(Poly3,Poly4,Poly).
multiply_polynomials([],P,[]).

binomial(Poly,1,Poly).

solve_polynomial_equation(PolyEquation,X,X = -B/A):-
        linear(PolyEquation), !,
        pad(PolyEquation,[p(A,1),p(B,0)]).
solve_polynomial_equation(PolyEquation,X,Solution):-
```

```
        quadratic(PolyEquation), !,
        pad(PolyEquation,[p(A,2),p(B,1),p(C,0)]),
        discriminant(A,B,C,Discriminant),
        root(X,A,B,C,Discriminant,Solution).

discriminant(A,B,C,D):- D:= B*B - 4*A*C.

root(X,A,B,C,0,X= -B/(2*A)).
root(X,A,B,C,D,X= (-B+sqrt(D))/(2*A)):- D > 0.
root(X,A,B,C,D,X= (-B-sqrt(D))/(2*A)):- D > 0.

pad([p(C,N)|Poly],[p(C,N)|Poly1]):- !,
      pad(Poly,Poly1).
pad(Poly,[p(0,N)|Poly1]):-
        pad(Poly,Poly1).
pad([],[]).



linear([p(Coeff,1)|Poly]).
quadratic([p(Coeff,2)|Poly]).

offenders(Equation,X,Offenders):-
        parse([Equation],X,Offenders1),
        remove_duplicates(Offenders1,Offenders).

homogenize(Equation,X,Offenders,Equation1,X1):-
        reduced_term(X,Offenders,Type,X1),
        rewrite(Offenders,Type,X1,Substitutions),
        substitute(Equation,Substitutions,Equation1).

reduced_term(X,Offenders,Type,X1):-
        classify(Offenders,X,Type),
        candidate(Type,Offenders,X,X1).

classify(Offenders,X,exponential):-
        exponential_offenders(Offenders,X).

exponential_offenders([A^B|Offs],X):-
        free_of(X,A), subterm(X,B), exponential_offenders(Offs,X).
exponential_offenders([],X).

candidate(exponential,Offenders,X,A^X):-
        base(Offenders,A), polynomial_exponents(Offenders,X).

base([A^B|Offs],A):- base(Offs,A).
base([],A).

polynomial_exponents([A^B|Offs],X):-
        is_polynomial(B,X), polynomial_exponents(Offs,X).
polynomial_exponents([],X).

substitute(A+B,Subs,NewA+NewB):- !,
         substitute(A,Subs,NewA), substitute(B,Subs,NewB).
substitute(A*B,Subs,NewA*NewB):- !,
         substitute(A,Subs,NewA), substitute(B,Subs,NewB).
substitute(A-B,Subs,NewA-NewB):- !,
         substitute(A,Subs,NewA), substitute(B,Subs,NewB).
substitute(A=B,Subs,NewA=NewB):- !,
         substitute(A,Subs,NewA), substitute(B,Subs,NewB).
substitute(A^B,Subs,NewA^B):- !,
        integer(B), substitute(A,Subs,NewA).
substitute(A,Subs,B):-
        member(A=B,Subs), !.
substitute(A,Subs,A).

rewrite([Off|Offs],Type,X1,[Off=Term|Rewrites]):-
        homog_axiom(Type,Off,X1,Term),
        rewrite(Offs,Type,X1,Rewrites).
rewrite([],Type,X,[]).

homog_axiom(exponential,A^(N*X),A^X,(A^X)^N).
homog_axiom(exponential,A^(-X),A^X,1/(A^X)).
homog_axiom(exponential,A^(X+B),A^X,A^B*A^X).

subterm(Term,Term).
```

```
subterm(Sub,Term):-
        compound(Term), functor(Term,F,N), subterm(N,Sub,Term).

member(X,[X|Y]).
member(X,[F|T]):-
   member(X,T).

subterm(N,Sub,Term):-
      arg(N,Term,Arg),
      subterm(Sub,Arg).
subterm(N,Sub,Term):-
        N > 0,
        N1:= N - 1,
        subterm(N1,Sub,Term).

position(Term,Term,[]):- !.
position(Sub,Term,Path):-
        compound(Term), functor(Term,F,N), position(N,Sub,Term,Path), !.

position(N,Sub,Term,[N|Path]):-
   arg(N,Term,Arg), position(Sub,Arg,Path).
position(N,Sub,Term,Path):-
   N > 1, N1:= N-1, position(N1,Sub,Term,Path).

parse([A+B|Y],X,L1):- !,
         parse([A,B|Y],X,L1).
parse([A-B|Y],X,L1):- !,
         parse([A,B|Y],X,L1).
parse([A=B|Y],X,L1):- !,
         parse([A,B|Y],X,L1).
parse([A*B|Y],X,L1):- !,
         parse([A,B|Y],X,L1).
parse([A^B|Y],X,L):-
        integer(B), !, parse([A|Y],X,L).
parse([A|Y],X,L):-
      free_of(X,A), !,
        parse(Y,X,L).
parse([A|Y],X,[A|L]):-
      subterm(X,A), !,
        parse(Y,X,L).
parse([],X,[]).

free_of(Subterm,Term):-
        occurrence(Subterm,Term,N), N=0.

single_occurrence(Subterm,Term).

occurrence(Term,Term,1):- !.
occurrence(Sub,Term,N):-
        compound(Term), !, functor(Term,F,M), occurrence(M,Sub,Term,0,N).
occurrence(Sub,Term,0).

occurrence(M,Sub,Term,N1,N2):-
        M > 0, !, arg(M,Term,Arg), occurrence(Sub,Arg,N), N3:= N+N1,
        M1:= M-1, occurrence(M1,Sub,Term,N3,N2).
occurrence(0,Sub,Term,N,N).

multiple([X1,X2|Xs]).

remove_duplicates([],[]).
remove_duplicates([X|Xs],Ys):-
        member(X,Xs), !,
        remove_duplicates(Xs,Ys).
remove_duplicates([X|Xs],[X|Ys]):-
        remove_duplicates(Xs,Ys).

test_press(X,Y):- equation(X,E,U), solve_equation(E,U,Y).

equation(X,Y,Z):- ground(X), ground(Y), ground(Z).
```